Testing Graph Databases with Synthesized Queries

Zijing Yin
ETH Zurich
Switzerland
zijing.yin@inf.ethz.ch

Si Liu
ETH Zurich
Switzerland
si.liu@inf.ethz.ch

David Basin
ETH Zurich
Switzerland
basin@inf.ethz.ch

Abstract

Graph databases (GDBs) are increasingly used in many applications. However, their advanced features make them prone to logic bugs. Despite recent advances in GDB testing, a common limitation of existing approaches is the lack of ground truth for their test oracles. This results in both incorrectly identified bugs and overlooked bugs.

We introduce GQS (Graph Query Synthesis), the first automated testing approach for detecting logic bugs in graph databases (GDBs) based on an established ground truth. GQS starts by randomly generating a graph and selecting a set of properties associated with its elements, whose key-value pairs form the expected result set serving as the ground truth. It then synthesizes a query intended to retrieve these values from the graph. When the query is executed on the graph by the GDB under test, any discrepancy between the actual result set and the ground truth indicates a logic bug. To extensively test a GDB, we develop novel techniques that synthesize both syntactically and semantically complex queries.

We implement GQS in a tool that incorporates the first Cypher query synthesizer specifically designed for testing GDBs. Overall, we find 36 previously unknown bugs across four production GDBs, of which 26 are logic bugs, with some remaining undetected for up to five years. Additionally, our tool demonstrates superior effectiveness in bug detection compared to the state-of-the-art testers.

CCS Concepts

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

1 Introduction

Graph databases (GDBs) are increasingly used across a variety of modern applications. Unlike traditional relational databases, GDBs are optimized for managing the storage and retrieval of graph-structured data with nodes (e.g., users), relations (e.g., friendships), and properties attached to them (e.g., names for users and "since when" for friendships). This makes GDBs particularly well suited for applications such as recommender systems, social networks, and data mining [6], where intricate relationships and patterns in large datasets must be rapidly discovered and analyzed.

Unfortunately, GDBs' advanced features make them prone to logic bugs, where they silently produce incorrect query results. Figure 1 presents an example of a bug discovered in FalkorDB that existed for four years. FalkorDB returns a result that is incorrect as it has a wrong value in the `a3` field. Such logic bugs are widely recognized as being more challenging to detect than database crashes or exceptions [19, 47, 54, 62].

Related Work on GDB Testing. A key challenge in automatically detecting logic bugs is to develop an effective test oracle for determining whether a GDB correctly answers a given query. Recent research has produced excellent testers [16, 19, 22, 33, 61, 62] that attempt to address this challenge. In line with the growing trend of using randomized testing to effectively uncover system-level bugs in general [23, 31] and relational databases in particular [20, 47, 54], these tools all generate random graph data and queries to exercise the GDB under test. To determine correctness, they rely on either differential [34] or metamorphic [9] test oracles. A common limitation of these efforts is the absence of ground truth for their test oracles, resulting in both false positives (incorrectly identified bugs) and false negatives (overlooked bugs).

Differential testers [16, 61] execute the same query on different GDBs or across different versions of the same GDB. Individual execution results serve as an oracle for each other, and any discrepancy suggests a bug. Consequently, bugs that arise from libraries shared by multiple database implementations under test may be missed, e.g., two query results are the same, but both are incorrect. In addition, false positives may occur when queries yield inconsistent yet expected results for different GDBs (due to their intended designs or implementations [35, 39]). For instance, the Grand tool [61] was recently found to report a large number of false alarms [22, 62].

Metamorphic testers [19, 22, 33, 62] avoid false positives by design. They rewrite the originally generated query into a set of semantically related new queries and then check if their outputs maintain the same semantic relation, e.g., whether one result set is equivalent to another [33], is a subset of another [19], or is a union of multiple other result sets [22, 62]. However, many bugs can still go unnoticed because these testers focus solely on the relationships between the answers to different queries, rather than establishing a ground truth for each individual query. For example, suppose the query Q' is derived from the query Q based on the subset relation, and their query result sets are R' and R, respectively. While $R' \subseteq R$ passes the test, both result sets could be incorrect.

Additionally, metamorphic testers rely on oracles that may constrain the complexity of test queries, which is desirable to trigger subtle logic bugs. For instance, GDBMeter [22] bases its test oracle on a three-valued logic, which can be used only to filter clauses like `WHERE` (in Cypher language [11]). Consequently, bugs not rooted in incorrect predicate handling are likely to be overlooked.

Our Approach. We introduce GQS (Graph Query Synthesis), the first automated testing approach for identifying logic bugs in GDBs using an established ground truth. GQS begins by randomly generating a graph and selecting a set of properties associated with its elements, whose key-value pairs constitute the *expected result set* serving as the ground truth. It then synthesizes a *complex* query aimed at retrieving these values from the graph. When the GDB under test executes this query on the initially generated graph, any discrepancy between the actual result set and the expected result set reveals a logic bug.

```
1 MATCH (n2)<-[r1]-(n0), (n3)-[r2]->(n4)-[r3]->(n5)
      WHERE r1.id=13
2 UNWIND [n5.k2 <> r3.id, false] as a1
3 WITH DISTINCT n2, r3, n3, n4, n5, endNode(r1) as a2, n0
4 MATCH (n2)<-[r4 :T10]-(n0), (n3)-[r5]->(n4)-[r6]->(n5)
      WHERE (((r6.k85)+(n2.k11)) ENDS WITH 'qIIcZH6h') AND
       ((n2.k9) = -1982025281) AND (n5.k2<=-881779936)
5 RETURN n2.id as a3, r6.id as a4
6 // expected result: {a3:1, a4:16}  ✓
7 // actual result: {a3:4, a4:16}
```

Figure 1: A logic bug found in FalkorDB. The full bug-triggering query in Cypher has various search patterns and nested expressions (underlined), as well as different types of clauses and cross-clause variable references (colored).

To effectively test GDBs and uncover their logic bugs, GQS synthesizes test queries that are both syntactically and semantically complex. These queries challenge GDBs beyond common user scenarios; while they may differ from typical usage patterns, they are highly effective for increasing the likelihood of uncovering hidden bugs related to query parsing, optimization, graph traversal, etc. We generate these queries across four dimensions: (i) incorporating diverse search patterns, (ii) generating deeply nested expressions within a query, (iii) establishing highly correlated data dependencies between different parts of the query, and (iv) utilizing a wide range of language features, including various clauses and functions.

The query shown in Figure 1 is synthesized by GQS, which uncovered a previously unknown logic bug in FalkorDB. This test query incorporates various search patterns and deeply nested expressions, as underlined. Additionally, it utilizes a variety of clause types, such as `MATCH`, `UNWIND`, and `WITH`, as well as functions like `endNode`, and exhibits complex cross-clause dependencies (e.g., the variable `n5` is referenced in four different clauses), as highlighted.

Synthesizing complex GDB queries like that in Figure 1 from an expected result set is, however, highly non-trivial. GQS addresses this challenge by synthesizing queries in a *stepwise* manner, leveraging a collection of novel techniques. Specifically, we leverage the *chaining* structure of graph queries to devise a synthesis plan for the given expected result set, breaking the synthesis task into individual steps, each corresponding to a specific clause in the final query. In particular, these steps involve synthesizing queries that go beyond the simple `MATCH–RETURN` structure, allowing for the integration of a wider range of clauses and functions. For instance, as shown in Figure 1, although the desired result includes only `n2.id` and `r6.id` (line 5), additional synthesis steps are involved, such as introducing more nodes and relations in the `MATCH` step (line 1), distributing lists in the `UNWIND` step (line 2), and projecting in the `WITH` step (line 3). This also facilitates creating rich data dependencies across clauses, e.g., the referenced variables in the same colors. To ensure the final query result remains consistent with the ground truth, GQS schedules these additional synthesis steps in a *pairwise* manner, for example by removing any extra nodes or properties introduced earlier.

Additionally, we devise two techniques to enhance the complexity of the synthesized clause at each individual step. First, through *pattern mutation*, we instantiate the clause with varying search patterns, such as the two `MATCH` steps (lines 1 and 4). Second, to examine a wide range of expression features, including various functions and operators, we construct *branching and nested* expressions embedded within the clause, e.g., the `WHERE` predicate and the `endNode` function (lines 4 and 3).

We implement our GQS approach in a tool that incorporates the first Cypher query synthesizer tailored for extensively testing GDBs for logic bugs. We chose Cypher as it is the most widely adopted, fully-specified, graph query language [40], and it is used by prominent GDBs like Neo4j [39] and Memgraph [35]. However, our approach is general and can be adapted to test GDBs using other query languages, as discussed in Section 7.

Contributions. Overall, we make the following contributions.

- At the conceptual level, we provide a novel approach to tackle the test-oracle problem of detecting logic bugs in GDBs by synthesizing test queries based on an established ground truth.
- At the technical level, we propose a stepwise synthesis approach called GQS, incorporating novel techniques that ensure the synthesized test queries are complex, thereby stressing GDBs.
- At the practical level, we realize GQS in an automated testing tool and assess it on four extensively tested production GDBs supporting Cypher queries: Neo4j, Memgraph, Kùzu [24], and FalkorDB [12]. Our tool discovers 36 new bugs, including 26 logic bugs, where some remained latent for up to five years. Compared to the state-of-the-art testers, our tool also exhibits superior effectiveness in bug detection.

We have open-sourced our approach and synthesizer to support further research on GDB testing and related applications.[1]

2 Background

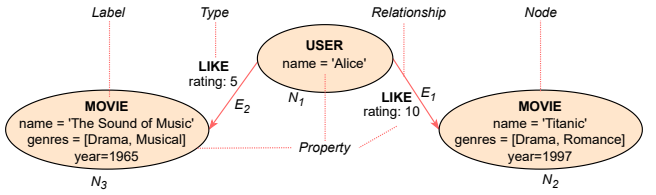
2.1 GDBs and Labeled Property Graphs

Graph databases (GDBs) structure data as a graph $G = \langle N, R \rangle$, where N denotes nodes (or objects) and R denotes relations (or relationships). The nodes and relations are also associated with labels or types, which define their categories. Furthermore, additional information may be associated with them via properties. Following the Cypher Reference [42], we define a property p as a key-value pair (k, v). The key k is a tuple $\langle e, n \rangle$, where $e \in (N \cup R)$ specifies the graph element, and $n \in P_n$ is the property name. Accordingly, $v \in V$ is the value. The sets P_n and V are domain dependent. This structure, known as a *labeled property graph* (LPG) [8], is widely used in GDBs such as Neo4j and Memgraph.

Example 2.1. Figure 2 illustrates an example of an LPG modeling users' movie preferences. Specifically, the node N_1 labeled `USER` is connected to two nodes, N_2 and N_3, labeled `MOVIE`, via relationships E_1 and E_2, both labeled `LIKE`. The nodes and relationships include properties such as $\langle N_2, genre \rangle$ with the value `[Drama, Romance]` and $\langle E_1, rating \rangle$ with the value `10`.

Additionally, we define the *expected result set*, denoted as $P = \{p_1, p_2, \ldots, p_n\}$, as a collection of properties attached to graph elements. This set serves as the ground truth for validating the query results. For instance, the two properties in the above example form an expected result set.

[1] Our repository is available at https://github.com/Graph-Query-Synthesis/GQS.




```
1 // Query 1:
2 MATCH (m:MOVIE) WHERE m.year=1997
3 RETURN m.name, m.year
4 // Query 2:
5 MATCH (p :USER)-[r :LIKE]->(m :MOVIE)
6     WHERE p.name = 'Alice' AND r.rating >= 8
7 UNWIND m.genres AS LikedGenre
8 WITH DISTINCT m.name as MovieName, m.year as Year
9 RETURN MovieName, Year
```

Figure 2: An example labeled property graph storing the movies that users like (top) and two examples of Cypher queries retrieving movie's name and year (bottom).

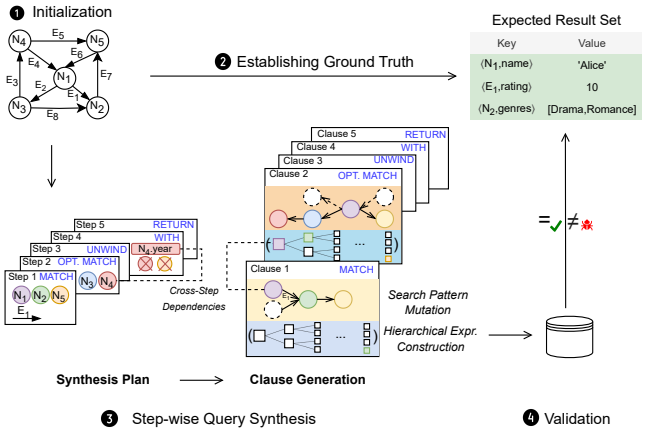


Figure 3: The workflow of GQS.

2.2 The Cypher Query Language

Cypher [40] is the *de facto* query language for GDBs. A Cypher query is structured as a sequence of *clauses*. Each clause takes as input a table of intermediate status (an empty table if it is the first clause), and produces a new table of intermediate status, serving as the input to the next clause. The last clause's output is the query result. Each clause can also refer to the LPG when required, for example, introducing graph elements to the table.

Cypher employs five main clauses to retrieve data, namely `MATCH`, `OPTIONAL MATCH`, `UNWIND`, `WITH`, and `RETURN`. The `MATCH` clause searches for the user-specified patterns in an LPG and introduces the matching elements into the intermediate result. `OPTIONAL MATCH` allows partial matching and fills in the missing parts of the pattern with *null* values. The `UNWIND` clause expands a list into individual rows, enabling subsequent clauses to perform operations on each element within the list. The `WITH` clause allows manipulating the intermediate result before passing it to the following clauses, for example, introducing new variables through projections or removing existing variables in the intermediate table by excluding them from the projection list. The `RETURN` clause has the same semantics as `WITH`, but it is used as the final clause in queries.

Additionally, the `UNION` and `CALL` clauses can be combined with the above clauses. The `UNION` clause connects two queries and computes the union of their outputs; the `CALL` clause invokes database engine procedures. Subclauses like `WHERE`, `ORDER BY`, `SKIP`, and `LIMIT` function as refinement operations within clauses. For instance, the `WHERE` clause adds constraints to the patterns described in the `MATCH` clause or filters the results in the case of `WITH`; the `ORDER BY` clause sorts results; the `LIMIT` and `SKIP` clauses truncate result sets. These subclauses mirror SQL functionalities.

Example 2.2. Figure 2 shows two Cypher queries that retrieve the same expected result set with keys $\langle N_2, name\rangle$ and $\langle N_2, year\rangle$. The first query employs a simple `MATCH`–`RETURN` structure. The second query returns the same result, but incorporates more complex features (aiming to exercise additional functionalities of query processing). The `MATCH` clause (line 5), together with the `WHERE` clause (line 6), searches for the movies liked by `Alice` and rated at least 8. Note that variables, such as `p` and `r`, bind to the nodes and relations, such as the nodes with `USER` label and relations with `LIKE` type, respectively. The `UNWIND` clause (line 7) then expands the list of genres associated with the matching movies into individual rows (two in this case: one row for `Drama` and the other row for `Romance`) under the column `LikedGenre`. The `WITH` clause (line 8) projects the node property like `name` as `MovieName` and removes duplicates with the `DISTINCT` operator. Finally, the `RETURN` clause (line 9) returns the query result.

Cypher also uses six additional clauses for writing data to the database, including `CREATE` (for creating graph elements), `SET` (for updating properties), `MERGE` (which acts as a combination of `MATCH` and `CREATE`, allowing conditional actions depending on whether the specified data already exists or is newly created), `DELETE`, `DETACH DELETE`, and `REMOVE` (for removing graph elements or properties). Unlike data retrieval clauses, which are utilized by test oracles to determine if the GDB under test correctly answers a given query, these clauses are mainly used for initializing or updating graphs.

Further details about these clauses can be found in the Cypher Query Language Reference [42].

3 Graph Query Synthesis

In this section, we present GQS (Graph Query Synthesis), an automated testing approach for detecting logic bugs in GDBs. GQS is unique in its ability to establish ground truth for complex graph queries that are used to test these databases extensively.

3.1 Overview

Figure 3 illustrates GQS's workflow, consisting of four main steps.

❶ **Initialization.** We begin by randomly generating a graph. For instance, the graph in Figure 3 comprises five nodes and eight relations. Additionally, labels and properties are randomly assigned to the nodes and relations, and indexes are created for these labels and properties. The graph is then loaded into the GDB under test.

❷ **Establishing Ground Truth.** We randomly select properties from graph elements to form an *expected result set*, such as $\langle N_1$, `name`$\rangle$, $\langle E_1$, `rating`$\rangle$, and their corresponding values, which are then stored in a table as key-value pairs. This serves as the ground

truth. Subsequently, we synthesize a graph query that is designed to produce these key-value pairs as the result upon execution.

❸ **Step-wise Query Synthesis.** We devise a synthesis plan consisting of multiple steps to obtain the expected result set. Each step focuses on synthesizing a part of the final query, such as the `MATCH` clause in Step 1 and the `RETURN` clause in the last step, as shown in Figure 3. To enhance the final query's complexity, these steps combine a variety of clauses, such as `WITH`, extending beyond the simple `MATCH–RETURN` structure (see Section 3.2 for details).

These clauses also introduce nodes, properties, and aliases outside the expected result set. To guarantee that the final query result still aligns with the set, we schedule these clauses in a pairwise manner. For instance, the extra red node N_4 added at Step 2 using `OPTIONAL MATCH` is removed at Step 4 using `WITH`. Across different steps, data dependencies are also established via variable references, as illustrated by the colors and dashed lines in the figure, leading to semantic connections between clauses (see Section 3.3 for details).

Based on the synthesis plan, concrete clauses are generated for each step. To increase their complexity, we mutate search patterns to explore a diverse set of query plans in the GDB engine, such as graph traversal processes, as illustrated by the dashed arrows and nodes in Clauses 1 and 2 (see Section 3.4). Furthermore, we embed hierarchical expressions within the generated clause, incorporating branching and nested structures, to examine GDB functions and operators (see Section 3.5). Finally, we chain all the clauses together to form the complete query.

❹ **Validation.** As the final step, we execute the synthesized query on the GDB under test. Any discrepancy between the actual execution result and the ground truth established at Step ❷ indicates the presence of a logic bug.

The above four steps illustrate one iteration of our testing process, which can be repeated. In subsequent iterations, we can either proceed with Step ❸ to synthesize another graph query, potentially with a different number of steps; continue with Step ❷ to generate new queries for other extracted properties; or restart with Step ❶ to create a new GDB, which may vary in size. These actions are chosen randomly. Before testing begins, we configure both the number of iterations and the number of synthesis steps per iteration.

At the core of our GQS approach is the synthesis of complex queries from a given expected result set (Step ❸). The remainder of this section provides a detailed explanation of this process.

3.2 Integrating Diverse Clauses

To synthesize a query whose result is the established ground truth, i.e., the expected result set, a simple `MATCH–RETURN` structure is sufficient, as shown in the first query of Figure 2. However, to extensively test the GDB, more complex queries are desirable. Therefore, GQS incorporates additional clauses during query synthesis, distributing them across steps. These additional clauses introduce diverse graph elements and aliases beyond the expected result set, to build more cross-step dependencies, enable different search patterns, etc. To ensure the final query still retrieves the desired properties, GQS schedules these clauses in a *pairwise* manner, e.g., by removing the extra nodes or properties added earlier.

Table 1: Paired operations for adding and removing elements and aliases, as well as expanding and truncating lists.

Notation	Operation	Clause
$\mathcal{E}^+$	introduce elements	(`OPTIONAL`) `MATCH`
$\mathcal{E}^-$	remove elements	`WITH`, `RETURN`
$\mathcal{A}^+$	create aliases	`WITH`, `RETURN`
$\mathcal{A}^-$	remove aliases	`WITH`, `RETURN`
$\mathcal{L}^+$	expand lists	`UNWIND`
$\mathcal{L}^-$	truncate lists	`WITH`, `RETURN`

Based on the semantics of the five main clauses related to data retrieval [42], we devise the following three pairs of "add and subtract" operations, as also illustrated in Table 1.

- **Introduce/Remove Elements.** Graph elements, such as nodes and properties, are introduced using (`OPTIONAL`) `MATCH` clauses. They can be removed using the `WITH` or the `RETURN` clause by excluding them from the intermediate or final result set, respectively.
- **Create/Remove Aliases.** Aliases can be bound to expressions, including those for property accesses (e.g., `n2.id` is bound to the alias `a3`, as shown in Figure 1), and to existing elements, serving as referenceable variables in the query. Both `WITH` and `RETURN` clauses can create or remove these aliases.
- **Expand/Truncate Lists.** Lists can be expanded into separate rows using the `UNWIND` clause. This duplicates the table of intermediate status based on the list's length, assigning each list element to one of the copies with a new alias. To truncate the expanded list and retain a single copy, the `WITH` or `RETURN` clause along with result refinement subclauses like `LIMIT` and `WHERE` can be applied.

To arrange these paired operations across individual steps, we classify them into two categories: (i) essential operations, which introduce properties in the expected result set, and (ii) supplementary operations, which are independent of the expected result set.

Specifically, for each property key $\langle \mathcal{E}, p \rangle$ in the expected result set, there are two types of essential operations. The first type introduces the graph element itself, denoted as $\mathcal{E}^+$, since accessing the property relies on the existence of the element. The second type refers to the actual access to the property, denoted as $(\mathcal{E}.p)^+$. Since the element $\mathcal{E}$ itself is not part of the ground truth, a paired removal, denoted as $\mathcal{E}^-$, is also scheduled.

Supplementary operations involve elements unrelated to the ground truth, including the introduction of extra graph elements, creation of aliases, and expansion of lists. All these operations are performed randomly: extra nodes or relations are selected at random from the graph; aliases are bound to randomly chosen graph elements or expressions (see Section 3.5 for expression generation); and list items are instantiated with similarly generated random expressions. As with Category (i), each of these operations is paired with a corresponding removal operation.

Example 3.1. Consider the initial graph in Figure 3 and the designated ground truth with property keys $\langle N_1, name \rangle$, $\langle E_1, rating \rangle$, and $\langle N_2, genres \rangle$. Category (i) involves nine essential operations, i.e., N_1^+, $(N_1.name)^+$, N_1^-, E_1^+, $(E_1.rating)^+$, E_1^-, N_2^+, $(N_2.genres)^+$,

and N_2^-, where three pairs of add-subtract operations are arranged for N_1, E_1, and N_2 that are not part of the expected result set.

Operations in Category (ii) are then supplemented by GQS. For example: an additional element N_3 can be added; an alias a bound to a property-access expression $N_4.year$ can be introduced; and an array l associated with the expression $[N_5.name]$ can also be expanded. The involved operations include N_3^+, a^+, N_4^+, N_5^+, and l^+, as well as their paired removals N_3^-, a^-, N_4^-, N_5^-, and l^-.

3.3 Building Cross-Step Dependencies

Our GQS approach distributes all operations across steps to build complex cross-step dependencies through referenceable variables in search patterns or expressions. However, *valid* dependencies are established only when operations are scheduled in the correct order. For instance, the property-access operation $(N_1.name)^+$ can only be assigned to a step after N_1 has been introduced in an earlier step; otherwise, the dependency cannot be formed as the query is syntactically invalid.

When scheduling operations, two key temporal constraints must be considered to build valid dependencies. First, in any task, the graph elements involved must have been introduced before they are referenced. Second, removal operations must be scheduled after their corresponding add operations.

We denote these temporal constraints as $O \prec O'$, meaning that the operation O must be scheduled before the operation O'. In the above example, the constraint $N_1^+ \prec (N_1.name)^+$ must be satisfied. Additionally, as Cypher allows element removal to be scheduled in the same clause as alias binding [42], we also account for weak constraints, denoted as $\preceq$. For example, $(N_1.name)^+ \preceq N_1^-$ means that the removal of the element N_1 can be scheduled either at the same step or after its property access $N_1.name$.

Example 3.2. Overall, the eight constraints for establishing valid dependencies in Example 3.1 are:

(1) $N_1^+ \prec (N_1.name)^+ \preceq N_1^-$
(2) $E_1^+ \prec (E_1.rating)^+ \preceq E_1^-$
(3) $N_2^+ \prec (N_2.genres)^+ \preceq N_2^-$
(4) $N_3^+ \prec N_3^-$
(5) $N_4^+ \prec a^+ \preceq N_4^-$
(6) $N_5^+ \prec l^+ \preceq N_5^-$
(7) $a^+ \prec a^-$
(8) $l^+ \prec l^-$

GQS schedules operations in different steps while ensuring no violation of constraints by adapting topological sorting. Specifically, it first scans the operations to be arranged, and then constructs a directed acyclic graph (DAG) representing the involved constraints, where each node is an operation and each edge is the temporal order between two operations. The distribution procedure takes as input the DAG and outputs, for each step, the assigned operations and the referenceable variables. These variables can then be referenced in search patterns or expressions, thus building data dependencies across steps. The pseudocode is given in Algorithm 1.

The procedure starts by traversing the DAG (line 4) and collects the nodes with zero indegree, i.e., operations that are ready to be assigned as they do not depend on any unassigned operations. It then checks whether the clause type of an operation o matches that of the operations already assigned to the current step, as each step corresponds to a single clause type. If both conditions are met, o is assigned to the current step based on a random decision (line 5).

Algorithm 1 Scheduling operations

Input: G, a DAG comprising operations as nodes and their constraints as edges
Output: *Step*, an array with the operations assigned to each step; *Var*, an array with the referenceable variables at each step

1: $i \leftarrow 1$ ▷ the current step
2: **while** G is not empty **do**
3: $Step[i] \leftarrow \emptyset$ ▷ initializing current step's ops
4: **for** o in G **do**
5: **if** $deg^-(o) = 0 \wedge$ align($Step[i]$, o) $\wedge$ rand() **then**
6: $Step[i]$.append(o)
7: **for** o' in $o.weak\_related$ **do** ▷ $o \preceq o'$
8: **if** $deg^-(o')$=1 $\wedge$ align($Step[i]$,o') $\wedge$ rand() **then**
9: $Step[i]$.append(o')
10: **end if**
11: **end for**
12: **end if**
13: **end for**
14: $Var[i] \leftarrow$ ref_vars($Var[i-1]$, $Step[i]$) ▷ $Var[0] = \emptyset$
15: G.remove($Step[i]$)
16: $i \leftarrow i + 1$
17: **end while**

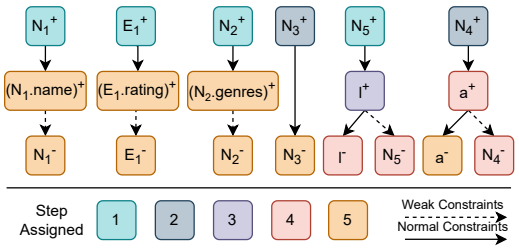


Figure 4: An illustration of assigning operations across different steps based on Example 3.2.

Example 3.3. Figure 4 illustrates the assignment of operations across steps, based on Example 3.2. For Step 1, six operations—N_1^+, E_1^+, N_2^+, N_3^+, N_4^+, and N_5^+—have zero indegree and share the same clause type `MATCH`. Among them, four operations are randomly selected and assigned to the first clause (colored in green).

The procedure then examines each operation o' that is weakly constrained by the assigned operation o, i.e., $o \preceq o'$ (line 7). If o' has no other constraints (i.e., its in-degree is one) and its clause type matches that of the operations already assigned to the current step, it may be included in the same step at random. (line 8)

Example 3.4. As shown in Figure 4, after assigning a^+ to Step 4 (colored in red), the operation N_4^- is also assigned to this step because it has only one constraint weakly related to a_4^+ and shares the same clause type `WITH`.

In each step, the set of referenceable variables is updated (line 14) by collecting those variables introduced earlier and excluding those removed in the current step. These variables, bound to aliases, nodes, and relations, can be referenced in search patterns and expressions (see Section 3.5). Subsequently, the assigned operations and their associated constraints are removed from the DAG (line 15). This process is repeated until the DAG becomes empty.

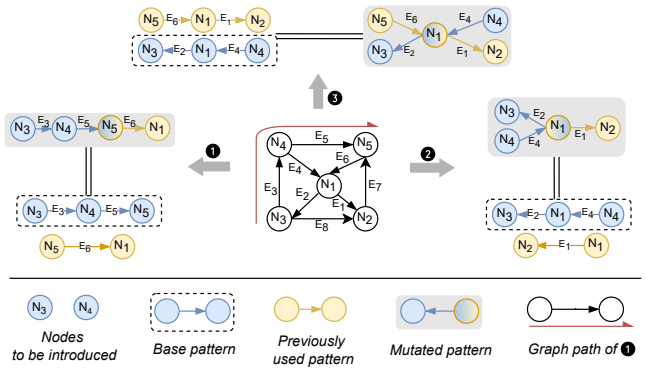


Figure 5: Three pattern mutation strategies.

Once the process terminates, operations are scheduled and clause types for each step are determined based on the mapping in Table 1. For example, Step 3 in Figure 4 will use `UNWIND` as its operation l^+ involves list expansion; Step 2 introduces elements ($\mathcal{E}^+$) and thus will employ `MATCH`, which may be randomly replaced with `OPTIONAL MATCH` for added variety. Next, we describe how GQS constructs each concrete clause to enhance query complexity.

3.4 Mutating Search Patterns

Diverse search patterns involve different ways of combining nodes and relations, as well as referencing elements introduced earlier in the query. Recall the search pattern `(p :USER)-[r :LIKE]->(m :MOVIE)` in Figure 2 (line 5). A "reverse" yet equivalent pattern, `(m :MOVIE)<-[r :LIKE]-(p :USER)`, would trigger different query plans, such as graph traversal processes, in many GDBs like Memgraph [35].[2] In the former pattern, all `USER` nodes are retrieved first, whereas, in the latter pattern, the traversal starts by retrieving all `MOVIE` nodes. Moreover, if a subsequent search pattern `(p)-[r2 :DISLIKE]->(m2 :MOVIE)` is used, the traversal will begin specifically from the `Alice` node, to which the variable `p` is bound through the first `MATCH` clause (line 5–6), to find the movies she dislikes.

To explore the diverse query-processing behaviors of GDBs, GQS introduces novel *pattern mutation* strategies that generate a large class of search patterns. These patterns both introduce new graph elements as specified in the plan and reference previously introduced ones. For instance, when synthesizing Clause 2 in Figure 3, the `MATCH` clause introduces nodes N_3 and N_4, while also referencing earlier elements such as N_1 and N_5.

GQS begins by collecting paths through the graph that contain the elements to be introduced. The sequences of nodes and relations along these paths form the *base patterns*. For example, the dashed boxes in Figure 5 show three base patterns, each containing N_3 and N_4, to be introduced. The path for the base pattern in Scenario ❶ is highlighted by the red arrow. GQS then iterates through all these base patterns and mutates them to new ones. First, for each base pattern, it identifies the graph elements that also appear in the patterns used in previous clauses (e.g., N_5 in Scenario ❶). There are three cases based on the shared element's position. For each, we devise a corresponding mutation strategy, which GQS applies to the base pattern, combining it with a previously used pattern.

- If the common element appears at the beginning or end of both the base and previous patterns, the two patterns are concatenated, as illustrated by ❶ in Figure 5. In this case, the mutated pattern now also references the previously introduced node N_1.
- When the common element is located at the start or end of either the base or the previous pattern (but not both), a branching mutation is applied, as depicted in ❷. In this scenario, the previously introduced node N_2 is incorporated into the mutated pattern, forming two branches that separately involve N_3 and N_4.
- If the common element is located in between the two ends of both the base and previous patterns, a cross mutation applies, as shown in ❸. The mutated pattern can then be split at the common element (N_1 in this case) into multiple subpatterns, which are then randomly recombined to form new search patterns.

Note that as both the base patterns and previously used patterns align with the graph structure, the mutated patterns derived from them naturally retain alignment to the graph, as illustrated by the three examples in Figure 5. Finally, we encode the mutated patterns into search patterns that are compatible with the `MATCH` clause. For example, the mutated pattern in Scenario ❷ can be encoded into two search patterns in a `MATCH` clause: `(n3)<-[e2]-(n1)-[e1]->(n2)` and `(n4)-[e4]->(n1)-[e1]->(n2)`.

During this process, GQS introduces additional mutations to enhance the complexity of the generated queries. Specifically, the labels or types of the corresponding nodes or relations are added to the search pattern element to enable index-based optimizations by the GDBs under test. Additionally, relations may either retain their original direction or disregard direction to trigger a wider range of graph traversal processes. For instance, the first search pattern mentioned above can be mutated into `(n3:L)-[e2]-(n1)-[e1]-(n2)`, where `L` specifies the label attached to `n3`, and the relation directions are removed. Furthermore, if the search pattern is part of an `OPTIONAL MATCH` clause, which allows matched subgraphs to partially align with the pattern, it can be extended with additional random patterns. This further increases the query's complexity.

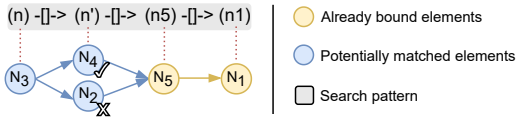


Figure 6: An illustrating example for predicate construction.

An encoded search pattern may match multiple subgraphs. For example, as shown in Figure 6, consider the pattern `(n)-[]->(n')-[]->(n5)-[]->(n1)` for Scenario ❶, where `n5` and `n1` were bound to the nodes N_5 and N_1 in previous steps. Both the subgraphs N_3-N_4-N_5-N_1 and N_3-N_2-N_5-N_1 match this pattern; however, only the former can correctly introduce the nodes N_3 and N_4, as scheduled (i.e., N_3^+ and N_4^+). GQS filters out undesired graph elements with predicates in the `WHERE` subclauses.

To construct such predicates, first, GQS scans through the search pattern and selects a pattern element that is already bound to a graph element. In our example, `(n5)` could be selected as it was bound to N_5 in the previous steps. If no such pattern elements exist, then one pattern element is randomly picked, and a predicate is constructed to ensure that it only matches the desired graph element.

[2] Some GDBs may optimize these two patterns into the same query plan. However, this is generally not the case for complex, yet semantically equivalent, search patterns [33].

Then, starting from this selected pattern element, GQS traverses the graph following the search pattern and constructs predicates when multiple matches could occur. In our example, starting from `(n5)`, GQS first checks the pattern segment `(n')-[]->(n5)`. Both N_4 and N_2 could match `(n')`, as they both have an outgoing relation connected to N_5. Therefore, a predicate, such as `n'.id = 4`, is constructed to ensure that `n'` is bound to N_4 (assuming identifiers are unique). This process is repeated until all the pattern segments are checked. Note that for `OPTIONAL MATCH` clauses, the randomly extended pattern segments are also checked to ensure that the pattern uniquely matches one subgraph, thereby avoiding an increase in the multiplicity of the results. Finally, the expressions used in these predicates will be further substituted with more complicated ones that also achieve the same filtering, as described in Section 3.5.

3.5 Generating Complex Expressions

While predicates with simple expressions, such as `n'.id = 4`, are sufficient to filter out undesired graph elements, more complex expressions are preferable to extensively test GDBs.

Previous work [16, 47] provides a good basis for generating complex expressions, like `char_length('abc')+sqrt(round(1.2))`, that satisfy specific value constraints, such as evaluating to 4 as in the right-hand side of the above predicate. GQS adapts the approach proposed in [16] to construct such expressions. Specifically, GQS first selects an expression template for the new expression. This template includes functions (e.g., `sqrt(par)`), operator applications (e.g., `par1 + par2`), and string match expressions (e.g., `par1 CONTAINS par2`), where `par1` and `par2` denote the parameters of the template. Such templates function as the skeleton of the expressions. Next, GQS converts the value constraint into respective sub-constraints for these parameters. For instance, if the template `par1 + par2` is selected, the sub-constraints for `par1` and `par2` can be evaluating to 3 and 1, respectively. This process repeats *recursively* to construct an expression for each parameter until the nesting depth is reached.

Replacing Property-Access Expressions. Compared to expressions like the above, generating complex expressions to replace property-access expressions in predicates, such as `n'.id`, is more challenging because the new expressions must still effectively filter out undesired graph elements. For example, the expression `sign(n'.id)` cannot replace `n'.id` as both `n'.id = 4` (i.e., N_4) and `n'.id = 2` (i.e., N_2) would evaluate to the same value 1. As a result, this expression is ineffective in ruling out N_2.

GQS generates complex expressions by recursively nesting templates based on the original property access while ensuring the distinguishability of elements at each nesting level. The pseudocode for this process is provided in Algorithm 2. This algorithm takes as input two sets of graph elements, S_1 and S_2, one of their common property name P whose value can distinguish these two sets, and the nesting depth D. It outputs an expression *Exp* that, when instantiated with elements from the respective sets, evaluates to distinct values, denoted as V_1 and V_2, respectively.

The algorithm begins by initializing the expression *Exp* with the original property-access expression (line 1). The evaluation results are then initialized by instantiating *Exp* with elements from the respective sets, S_1 and S_2 (lines 2–3). Next, the algorithm determines

Algorithm 2 Replacing property-access expressions

Input: S_1 and S_2, two graph element sets; P, a common property name; D, the nesting depth
Output: *Exp*, an expression that, when instantiated with S_1 and S_2, evaluates to different values with respect to P

```
1:  Exp ← original_property_access_expr(P)
2:  V1 ← property_val(S1, P)          ▷ instantiating Exp with S1
3:  V2 ← property_val(S2, P)          ▷ instantiating Exp with S2
4:  Type ← get_type(Exp)
5:  while D > 0 do
6:      Tplt ← exp_template_with_param_type(Type)
7:      New_Exp ← Tplt.nest(Exp)
8:      if New_Exp.eval(V1) ∩ New_Exp.eval(V2) = ∅ then
9:          V1 ← New_Exp.evaluate(V1)
10:         V2 ← New_Exp.evaluate(V2)
11:         Exp ← New_Exp
12:         Type ← get_type(Exp)
13:     end if
14:     D ← D − 1
15: end while
```

Exp's data type and selects a new expression template that includes a parameter matching this data type (lines 4–6). The new expression is instantiated with the original *Exp* as its parameter (line 7). If the template requires additional parameters, they are filled randomly. The evaluation results for the new expression are then calculated for both sets (line 8). If the evaluation results overlap, indicating the template cannot differentiate between the elements of S_1 and S_2, the algorithm selects a new template. Otherwise, *Exp* is updated with the new expression (line 11), and its data type is updated based on the result of the nested expression (line 12).

This process repeats, with the new *Exp* serving as a parameter for another expression template until the specified depth D is reached.

Example 3.5. In the aforementioned example, S_1 and S_2 include N_4 and N_2, respectively, while P represents the property name `id`. GQS first finds an expression template that can accept integers as parameters (i.e., `n'.id`'s type). If `left(m.name, n'.id)` is chosen as the new expression template, GQS checks if it still evaluates to different values when instantiated with `n'.id = 4` and `n'.id = 2`, respectively. When this is the case, GQS then searches for another template that can use `left(m.name, n'.id)` as a parameter and remains distinguishable for `n'.id = 4` and `n'.id = 2`. The process continues until the predefined nesting depth is reached.

4 Implementation

We implement our GQS approach in an automated testing tool for GDBs that support Cypher queries. The overall codebase consists of 34K lines of Java code.

Integrating Different GDBs. GQS can be easily adapted to test GDBs. We have integrated GQS into four popular GDBs, i.e., Neo4j, Memgraph, FalkorDB, and Kùzu, using their Java drivers. Each of the first three GDBs requires around 100 LoC to configure and ensure stable testing. Kùzu requires the database schema information before initializing a random graph, and 150 LoC were needed to

meet this requirement. For testing new GDBs, GQS offers interfaces that allow for easy integration of their Java drivers.

Note that using an expected result set as the ground truth further facilitates the integration, as properties (or key-value pairs) provide a unified format widely supported by diverse GDBs. Moreover, this design choice does not compromise the effectiveness of GQS's bug detection: errors not directly related to graph properties, such as incorrect subgraph retrieval, still manifest as incorrect property values, e.g., erroneous outputs of element identifiers.

Supported Cypher Features. Our implementation covers all 11 clauses and subclauses for data retrieval, as documented in the Cypher Query Language Reference [42]; see also Section 5.3. The only exception is the `MANDATORY MATCH` clause, which is not supported by the four Cypher databases we test (Section 5.1).

All supported data retrieval clauses are synthesized according to the plan described in Section 3.2, except `UNION` and `CALL`. These two clauses are handled separately due to their unique semantics and varied implementations in practice. Specifically, since `UNION` connects two queries, it is added after the synthesis of two separate queries. The `CALL` clause invokes database engine procedures. However, some implementations intentionally deviate from the Language Reference [42]. For example, to retrieve graph element labels, Neo4j and FalkorDB provide the procedure `CALL db.labels()`, which align with the reference, while Kùzu and Memgraph do not provide such functionality. In addition, our graph initializer incorporates all six clauses for writing data to the database. Unlike the data retrieval clauses, these clauses are not used in our ground-truth validation but are instead employed to create and update graphs.

GQS supports an extensive library of 61 functions, as well as aggregation operators, property access operators, and mathematical operators, which are commonly supported by the four tested GDBs. Functions limited to only a few databases are excluded to ensure compatible queries. Subquery and transaction features are currently unsupported. Variable-length patterns are not supported as they could introduce undesired subgraph matches.

Handling GDB-specific Cypher Variations. By default, our implementation synthesizes Cypher queries in line with the Cypher Query Language Reference [42]. However, we also account for variations in how different GDBs implement certain Cypher features. For instance, Kùzu and FalkorDB deviate from the relation uniqueness requirement, allowing the same relation to be matched multiple times within a single query pattern. In the search pattern `(n1)-[e1]-(n2)-[e2]-(n3)`, `e1` and `e2` are expected to correspond to different relations in the graph according to the reference. However, in FalkorDB and Kùzu, the same relation can be matched by both `e1` and `e2`. To address this, we introduce additional `WHERE` predicates to filter out duplicate matches, e.g., `WHERE e1!=e2`.

5 Experiments

In this section, we conduct an extensive assessment of GQS, along with the state-of-the-art logic bug detectors for GDBs. We seek to answer the following questions:

Q1. Can GQS detect new bugs in production GDBs (Section 5.2)?
Q2. How do our key design choices contribute to GQS's effectiveness in bug detection (Section 5.3)?
Q3. Is GQS more effective in finding bugs than the state-of-the-art (Section 5.4)?

Table 2: Summary of the tested GDBs.

GDB	GitHub stars	Initial release	Tested version	LoC
Neo4j	13.2K	2007	5.18, 5.20, 5.21.2	1.4M
Memgraph	2.4K	2017	2.13, 2.14.1, 2.15, 2.17	0.2M
Kùzu	1.3K	2022	0.4.2, 0.7.1	11.9M
FalkorDB*	651	2023	4.2.0	2.8M

∗ RedisGraph, the predecessor of FalkorDB, has received 2K GitHub stars since its initial release in 2018.

Table 3: Summary of the bugs detected by GQS.

GDB	Logic bugs			Other bugs		
	#detected	#confirmed	#fixed	#detected	#confirmed	#fixed
Neo4j	2	2	2	3	3	3
Memgraph	6	6	1	1	1	0
Kùzu	5	5	5	2	2	2
FalkorDB	13	4	0	4	2	1
Total	26	17	8	10	8	6

5.1 Experimental Setup

We examine recent releases of four popular and heavily tested GDBs: Neo4j [39], the market leader with over two decades of development; Memgraph [35], an in-memory GDB specializing in real-time analytics; FalkorDB [12], the fork of RedisGraph [44], designed for large language models; and Kùzu [24], an emerging embeddable GDB. During our testing, when new versions of the GDBs were released, we deployed new instances to assess them. Table 2 provides details on the tested GDBs. Note that when comparing GQS with the state-of-the-art tools, we also test early releases of Neo4j, Memgraph, and FalkorDB (in this case, RedisGraph), which are not included in Table 2. These releases have been extensively tested using these competing tools; see Section 5.4 for details.

We initialize the tested GDBs using random graphs of varying sizes, with a maximum of 13 nodes and 500 relations. In addition, we set GQS to perform up to 9 synthesis steps. The maximum size of an expected result set is limited to 6. While larger parameter settings, such as larger graphs and expected result sets, may expose additional bugs, they also result in greater overhead for GQS. Hence, there is no single "best" parameter configuration. In practice, users should configure GQS based on their specific needs. However, throughout our testing, we observed that small to medium graph sizes and expected result sets are sufficient to uncover a large number of bugs. Specifically, all detected bugs were triggered on graphs with fewer than 12 nodes and 31 relations, and involved fewer than 5 properties in the expected result sets; detailed distributions of bugs across these parameters are available in our repository. Overall, we recommend that users start with small parameter settings and gradually increase them during the testing process.

Our experiments were conducted on a Linux workstation running Ubuntu 22.04, with AMD Ryzen 9 7950X and 128GB of memory.

5.2 Discovering New Bugs

We have detected and reported 36 previously unknown bugs across all four GDBs, as summarized in Table 3. For more details, please refer to our repository. Among these bugs, 26 are logic errors that

yield incorrect query results. Additionally, there are 10 other bugs, including memory corruption, crashes, or unexpected exceptions, which can cause the GDBs to hang indefinitely, consume excessive memory, etc. As of this writing, the developers have confirmed 25 issues, with 14 resolved in the latest releases. Notably, almost all bugs identified in Neo4j, Memgraph, and Kùzu have been confirmed, and the majority have been fixed, following our reports. Due to the relatively longer response time from the FalkorDB developers, the number of confirmed bugs remains limited at the time of this paper's publication. However, the confirmations we received from the other three GDBs give us strong confidence that GQS is effective and that the bugs identified in FalkorDB are also genuine.

Interesting Bugs. We presented a representative bug in Section 1. In the following, we provide examples of three other bugs.

Example 5.1. Figure 7 illustrates a Cypher query that triggers a logic bug in Neo4j. This query consists of four main steps, with two `MATCH` clauses searching for the desired patterns and an `UNWIND` clause in between expanding the arrays involved. Instead of returning the correct value of `r4.k191` as specified in the query (line 4), Neo4j incorrectly returns the property value of a node (`n4`) that is not even queried for. Such bugs could result in data leakage or, if exploited, unauthorized data access.

```
1 MATCH (n0 :L11)<-[r0 :T3]-(n1) WHERE (((NOT (NOT ...
2 UNWIND [(r0.k186), 557243387] AS a0
3 MATCH (n2 :L11 :L5)-[r1 :T3]->(n3 :L11), ..., (n7 :L11 :
      L5)-[r4 :T3]->(n8 :L11 :L5 :L4) WHERE ...
4 RETURN (r4.k190) AS a3, (r4.k191) AS a4
5 // expected result: 6 rows of {a3:v6z5e, a4:true} ✓
6 // actual result: 6 rows of {a3:v6z5e, a4:WEJ6Mifdo} 🐞
```

Figure 7: A logic bug found in Neo4j: an incorrect return for `r4.k191`'s value. Parts of the query are omitted for simplicity.

Example 5.2. Figure 8 presents a test query that causes Memgraph to incorrectly produce an empty result. The query involves five steps and eight clauses and subclauses, which collectively trigger Memgraph's complex optimization logic. The bug arises from an unexpected optimization combination in the query plan, specifically Cartesian product optimizations combined with filtering.

Fixing this bug proved challenging. Despite quickly initiating an investigation, the developers' progress stalled after one month. As a temporary solution, they recommended disabling Cartesian product optimizations, which compromised performance to ensure correctness. A fix was finally implemented after six months.

While our focus is on logic bugs, GQS can also trigger memory corruption bugs, unexpected exceptions, or GDB crashes. The test oracles for identifying these issues come at no additional cost.

Example 5.3. Figure 9 shows a query that exposes a memory leak in the latest release of Memgraph. This issue originates from the `replace` function, which is currently underspecified for handling empty strings. Consequently, when the query attempts to match an empty string in the string "`tS15G`" and replaces it with another string "`U1lsWFvRw`", Memgraph hangs indefinitely and consumes over 50GB of memory within five minutes.

This bug is severe. It can significantly affect GDB availability and lead to security vulnerabilities like denial-of-service attacks.

```
1 MATCH (n0 :L0 :L6 :L11)<-[r0 :T2]-(n1), (n2 :L6)<-[r1 :T2
      ]-(n3 :L0) WHERE ...
2 UNWIND [-1465465557] AS a0
3 MATCH (n4 :L0)<-[r2 :T2]-(n5 :L0 :L6) WHERE ...
4 UNWIND [(n0.k65)] AS a1
5 RETURN (r1.k86) AS a2, (n3.k4) AS a3, (r1.k87) AS a4
      ORDER BY a4 DESC
6 // expected result: {a2:0spkB,a3:false,a4:SqpUzADY6} ✓
7 // actual result: {} 🐞
```

Figure 8: A logic bug in Memgraph caused by unexpected optimization combinations.

Notably, it had persisted for over three years since Memgraph's earliest publicly accessible release, before being discovered by GQS.

```
1 WITH replace('tS15G', '', 'U1lsWFvRw') AS a0
2 RETURN a0
3 // expected result: {a0:'tS15G'} (empty string ignored)
      or a parse error (empty string disallowed) ✓
4 // actual result: Memgraph hangs, consuming memory 🐞
```

Figure 9: A memory leak uncovered in Memgraph due to underspecified handling of empty strings.

In addition to the bug severity discussed above, we reviewed Kùzu's developer discussions and patches to further assess the impact of our detected bugs, based on their detailed bug reports. Among the seven bugs identified, two are not only logic bugs leading to incorrect outputs, but they also involve unsafe type usage, potentially resulting in memory corruption. Such issues could have security implications for applications built on top of Kùzu.

Notably, many bugs (triggered by our complex test queries) are located in core, frequently invoked functions. This suggests that the processing of queries in typical user scenarios could also be affected by the same underlying issues. For example, one bug in Kùzu stems from an error in a common helper routine for binary operators, potentially impacting all queries relying on this functionality.

5.3 A Closer Look at GQS

We have demonstrated GQS's effectiveness in bug detection. To further assess the contributions of our key design choices to the bugs found, we analyze all 36 bug-triggering test queries with respect to five aspects: synthesis steps, Cypher features, data dependencies, search patterns, and nested expressions.

Overall, the majority of the detected bugs involve many steps, diverse search patterns, deeply nested expressions, rich clause types, and complex cross-clause references within the test queries. This underscores the importance of complex queries in effectively triggering bugs, which is the focus of our GQS approach.

Synthesis Steps. GQS generates queries in a stepwise manner. Figure 10 shows the distribution of all detected bugs across the four tested GDBs, categorized by the different synthesis steps involved. As we can see, 80% of the bugs are triggered by queries synthesized with at least three steps, with those composed of four to six steps being particularly effective. This result is consistent with the expectation that more complex queries are more likely to cover corner cases in, e.g., pattern matching (Example 5.1) and combining optimizations (Example 5.2). In contrast, existing metamorphic testers often use only one or two clauses, potentially missing bugs.

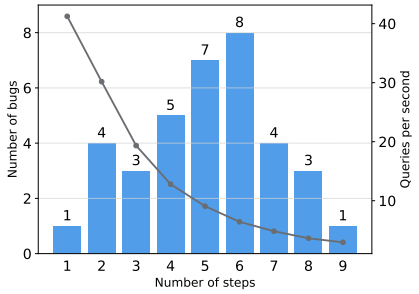


Figure 10: Distribution of all 36 bugs across the tested databases, categorized by the different synthesis steps involved.

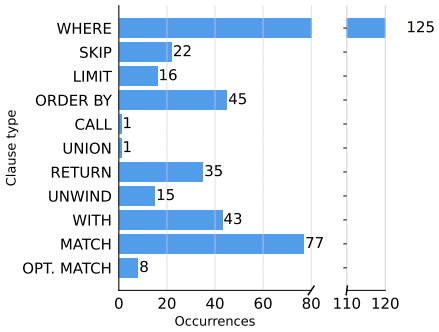


Figure 11: The aggregated number of clauses involved in the bug-triggering test queries.

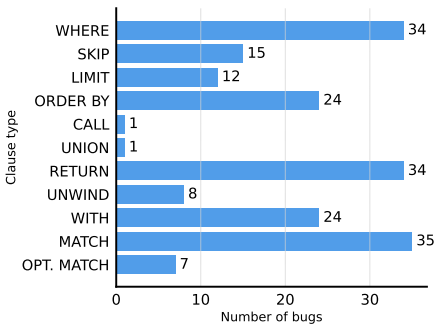


Figure 12: The number of bugs related to different types of clauses.

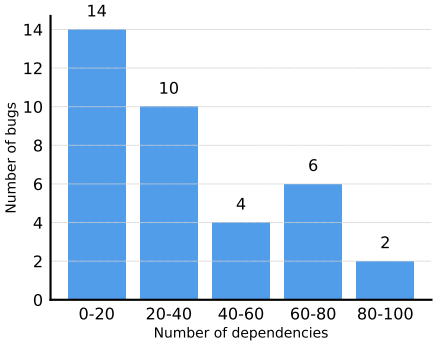


Figure 13: Distribution of bugs based on the number of dependencies involved.

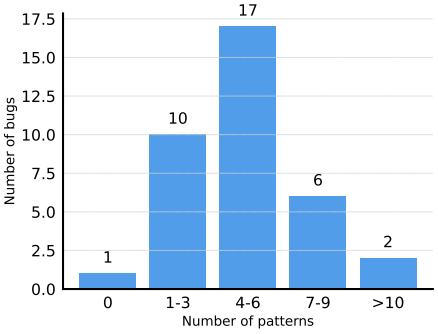


Figure 14: Distribution of bugs based on the number of patterns involved.

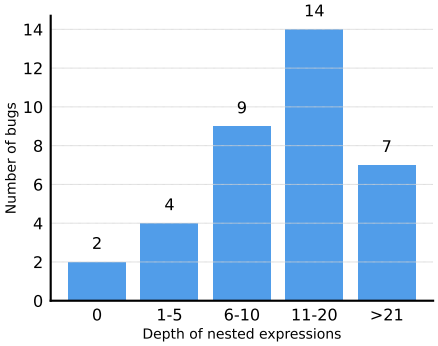


Figure 15: Distribution of bugs based on the depth of nested expressions.

However, test queries with fewer steps still account for a non-negligible number of bugs. These queries are often sufficient to expose errors in functions or operators, as illustrated in Example 5.3.

More steps necessitate additional synthesis and execution time, as confirmed by our analysis presented in Figure 10. Specifically, as the number of steps increases, the number of test queries completed per second decreases. For example, queries synthesized with nine steps require 6.6 times more execution time on average compared to those synthesized with only three steps. Excessively increasing the steps reduces the bug-finding performance. In practice, users of GQS can configure the number of synthesis steps to strike a balance between effectiveness and efficiency in bug detection. Additionally, this configuration should take into account the type of GDB under test. For example, in-memory GDBs such as Memgraph can typically handle test queries with more synthesis steps, processing around six queries per second with nine synthesis steps; whereas on-disk GDBs like Neo4j handle only about three queries per second for the same number of steps, likely due to higher I/O overhead.

Cypher Features. GQS-synthesized queries encompass a wide range of Cypher features, including various clauses. As indicated by our analysis, all of these clauses are involved in the bug-triggering test queries. Note that our analysis excludes data manipulation clauses like `CREATE` and `SET` as they are not relevant to data querying (or ground truth validation).

A clause can occur multiple times per query due to the random assignment at each synthesis step. As shown in Figure 11, the `MATCH` clause appears frequently, with over 70 aggregated occurrences across all the queries. This is expected because pattern matching underlies GDB querying. Note that the `WHERE` clause appears more than 100 times as it serves as the filtering subclause for both `MATCH` and `WITH`. Figure 12 presents the number of bugs related to different clauses. Most of the bugs involve the canonical Cypher skeleton `MATCH–WHERE–RETURN`; however, the number of bugs associated with other clauses is also significant. For example, 24 out of 36 bugs are triggered by queries containing the `ORDER BY` or `WITH` clauses.

In addition, a substantial number of Cypher functions, 32 in total, are involved in the reported bugs. See our repository for details.

Cross-Step Dependencies. During the synthesis of test queries, GQS establishes complex data dependencies across multiple steps. Figure 13 shows that, although a significant number of bugs involve fewer than 20 dependencies, over 61% of bugs are triggered by test queries with more than 20 dependencies. These dependency-heavy queries often lead to problematic execution paths in query planners during optimization, resulting in incorrect query outcomes.

Search Patterns and Nested Expressions. We also analyze the variety of search patterns used and the depth of nested expressions involved in the bug-triggering test queries. Figure 14 presents the distribution of bugs based on the number of patterns used. Notably, two-thirds of the bugs are triggered by test queries containing more than three distinct patterns.

Nested expressions also play a crucial role in triggering bugs. As shown in Figure 15, 83% of the bugs are associated with queries that contain more than five levels of nested expressions.

Table 4: Bugs missed by existing testers and their latencies. GDBMeter, Gamera, and GQT did not support Memgraph.

Tester	Neo4j	Memgraph	RedisGraph*	**Total**
GDsmith	1	2	15	18
GDBMeter	1	–	15	16
Gamera	2	–	16	18
GQT	1	–	16	17
GRev	2	2	16	20
Avg. latency (yrs)	2.2	3.4	4.0	–
Max. latency (yrs)	2.7	3.4	5.0	–

∗ All tools tested earlier versions of FalkorDB, i.e., RedisGraph.

Table 5: Comparison on test query complexity.

Tester	**Pattern**	**Expression**	**Clause**	**Dependency**
GDsmith	4.96	3.68	6.39	21.75
GDBMeter	0.86	2.24	1.94	1.97
Gamera	0.83	1.39	1.92	1.89
GQT	1.03	2.87	3.39	3.43
GRev	6.69	5.26	6.49	28.41
GQS	8.14	7.82	6.50	56.02

5.4 Comparison with State-of-the-Art

We compare GQS with five state-of-the-art logic bug detectors.

- The metamorphic bug detectors, GDBMeter [22], Gamera [62], GQT [19], and GRev [33], each applying specifically designed query rewrite rules to identify discrepancies between the result of the original query and that of the rewritten query.
- The differential tester GDsmith [16], which detects logic bugs by comparing the results of the same query across multiple GDBs.

5.4.1 Bug Latency. In line with common practice [21, 33], we analyze the latencies of the bugs found by GQS, i.e., the time span between the earliest version of the GDB in which a bug was introduced and when it was first discovered by us. This analysis provides a perspective on GQS's effectiveness in uncovering hidden bugs: if a bug exists in versions that predate those tested by existing tools, it suggests that the bug remained latent despite their extensive testing efforts. Our analysis excludes Kùzu as it is a relatively new GDB that has not yet been tested by existing tools. This results in a total of 29 bugs across the remaining three GDBs.

As shown in Table 4, over half of the 29 bugs reported by our tool were introduced in versions prior to those tested by the state-of-the-art. In particular, nearly all of the 17 bugs found in FalkorDB (see Table 2) were already present in RedisGraph versions predating those assessed by these tools. The average latency for these missed bugs ranges between two and four years, where the longest latent bug was introduced five years ago.

5.4.2 Test Query Complexity. Generating complex queries is essential for extensively-testing GDBs and triggering logic bugs, as we have already seen in Section 5.3. Therefore, we compare the queries generated by GQS and existing tools with respect to four aspects: patterns, expressions, clauses, and dependencies.

We randomly select 10,000 test queries generated by each tool and use the Cypher query parser [27] to convert them into abstract syntax trees [4]. We then iterate through these trees to collect the following metrics per query: (i) the number of patterns involved, (ii) the maximum depth of nested expressions, (iii) the number of clauses involved, and (iv) the number of cross-clause data references. We report the average for each metric, as shown in Table 5.

Overall, the test queries synthesized by our tool GQS exhibit substantially greater complexity than those generated by existing tools. In particular, each GQS-generated query utilizes an average of 8.14 patterns, which is approximately 180% more than the average of the other tools. GQS's queries also reach a maximum depth of 7.82 for nested expressions, around 50% deeper than the second-best tool GRev. Moreover, GQS demonstrates significantly higher complexity in data dependencies, with around 100% more cross-clause references compared to GRev. Note that the queries generated by GDBMeter, Gamera, and GQT are significantly less complex. This is partially due to the constraints imposed by their metamorphic oracles, which require test generation to adhere to predefined rewrite rules.

5.4.3 Test Oracle Effectiveness. While GQS is capable of generating more complex queries than existing tools, for a fair comparison of test oracles, we integrate our bug-triggering test queries with their respective oracles. Our evaluation focuses solely on whether these tools can identify the bugs when provided with the same test queries. Specifically, for metamorphic testers, we apply their rewriting rules to these queries and check whether the transformed queries reveal violations of metamorphic relations. For the differential tester GDsmith, we execute the queries across different GDBs and check for any discrepancies.

Missed Bugs. Overall, metamorphic oracles are insensitive to certain types of bugs, especially when their root causes are unrelated to the predefined metamorphic relations. When comparing with GDBMeter and GRev, we are able to quantify the number of missed bugs because GDBMeter's metamorphic relations are straightforward to implement, and GRev provides direct interfaces for applying its rewrite rules to given queries. Out of 26 logic bugs detected by GQS, GDBMeter and GRev could only identify 11 and 3, respectively, even when using the bug-triggering test cases.

Example 5.4. Figure 16 shows an example bug found by GQS that is missed by GDBMeter. When running Memgraph with the GQS-synthesized test query (lines 2–3), it produces an incorrect empty output, instead of the expected result (line 14). Upon applying GDBMeter's ternary logic partitioning [22] to the test query, all three partitioned queries (lines 5–13) also result in empty results. The union of these results matches the incorrect output (namely GDBMeter's oracle), causing GDBMeter to mistakenly pass the test. The key factor triggering this bug lies in the `WITH` clause, which is not accounted for by GDBMeter's metamorphic rules.

Unlike the above two metamorphic testers, Gamera and GQT do not provide interfaces for directly applying their metamorphic relations to specific queries. In addition, their query rewrites incorporate a degree of randomness. For instance, GQT randomly adds labels to nodes during query rewrites, resulting in infinitely many transformations. Consequently, it is infeasible to precisely quantify the number of bugs they might overlook. Therefore, we manually analyze their predefined metamorphic relations and observe that these tools are likely to miss some bugs reported by our tool, as their root causes are unrelated to the relations.

```
// GQS-synthesized test query
MATCH (n0)-[r0]->(n1) WITH r0 ...
WHERE ("1"<>n0.k99) RETURN r0.id AS a0

// GDBMeter-rewritten  query #1 (NOT)
MATCH (n0)-[r0]->(n1) WITH r0 ...
WHERE NOT ("1"<>n0.k99) RETURN r0.id AS a0
// GDBMeter-rewritten query #2 (IS NULL)
MATCH (n0)-[r0]->(n1) WITH r0 ...
WHERE ("1"<>n0.k99) IS NULL RETURN r0.id AS a0
// GDBMeter-rewritten query #3 (TRUE)
MATCH (n0)-[r0]->(n1) WITH r0 ...
WHERE TRUE RETURN r0.id AS a0
// expected result: {a0:20} ✓
// actual result (serving as GDBMeter's oracle): {} 🐞
// GDBMeter: {}∪{}∪{}={} (matching the incorrect oracle)
```

Figure 16: A Memgraph bug found by GQS that cannot be detected by applying GDBMeter's metamorphic rules.

Example 5.5. Figure 17 illustrates a representative bug found in the latest version of FalkorDB, which is unlikely to be detected by Gamera or GQT. This bug's root cause lies in FalkorDB's mishandling of UNWIND's semantics, resulting in some records not being fetched. Gamera's and GQT's oracles are unlikely to identify this bug because their metamorphic rules do not cover UNWIND.

```
UNWIND [1,2,3] AS a0
MATCH (n2 :L12)-[r1]-(n3) WHERE (((r1.id) = 13) AND ...
RETURN a0
// expected result: [{a0:1},{a0:2},{a0:3}] ✓
// actual result: [{a0:1}] 🐞
```

Figure 17: A bug found by GQS in FalkorDB's latest version that is unlikely to be detected by Gamera or GQT.

False Alarms. For the differential tester GDsmith, feeding it with GQS-generated bug-triggering queries resulted in no missed bugs. This is expected since our tested GDBs do not share a substantial common codebase; therefore, the same bugs do not exist across these GDBs. However, we observe a significant number of false positives, which may limit GDsmith's practical applicability. In particular, running GDsmith for 24 hours on the latest versions of Neo4j and Memgraph resulted in 1192 bug reports, of which 1160 are false positives, yielding an approximate 98% false positive rate. Such high false alarm rates mainly come from exceptions raised by invalid queries and output format differences.

5.4.4 24-Hour Empirical Testing. What about the bug detection effectiveness of each tool on its own? To answer this, we conducted a 24-hour empirical analysis of all testers using their respective query generators. As shown in Table 6, GQS demonstrates superior effectiveness, identifying a total of 13 bugs, including 11 logic bugs, which are significantly more than the other tools. Note that Table 3 (Section 5.2) reports a greater number of bugs than mentioned here, as our overall testing process spanned several months.

We also analyze how the number of bugs detected with each tool accumulates over the 24-hour period, as shown in Figure 18. Since Memgraph and Kùzu are either unsupported or not tested by all testers, we report results only for Neo4j and FalkorDB. Compared to the other testers, GQS offers two key advantages: it not only detects the largest number of bugs overall, but it also continues to trigger bugs consistently throughout the testing period. This highlights GQS's effectiveness in rapidly and reliably identifying bugs.

Table 6: Bugs detected over a 24-hour testing.

Tester	Neo4j	Memgraph	FalkorDB	**Total**
GDsmith	1 (1)	2 (1)	4 (4)	7 (6)
GDBMeter	1 (0)	–	1 (0)	2 (0)
Gamera	0 (0)	–	1 (0)	1 (0)
GQT	0 (0)	–	5 (5)	5 (5)
GRev	0 (0)	1 (1)	1 (1)	2 (2)
GQS	2 (1)	3 (2)	8 (8)	13 (11)

X (Y): among the X bugs found in total, Y bugs are logic bugs.

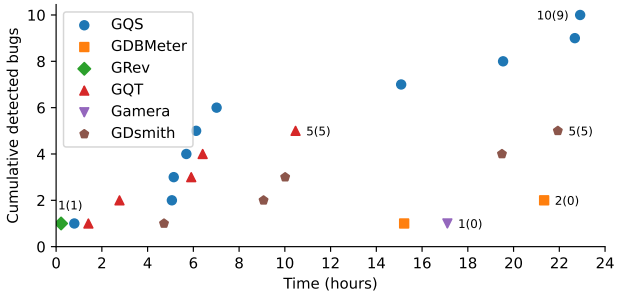


Figure 18: Cumulative bugs detected in Neo4j and FalkorDB over 24 hours. We follow the same setup as in Table 6.

While GQS can uncover hidden bugs missed by state-of-the-art tools (see Section 5.4.1), it may conversely miss bugs found by others. In our 24-hour experiment, GQS failed to detect two crash bugs in FalkorDB that were identified by GDBMeter and Gamera after 21 and 17 hours, respectively. This is likely because both testers maintain a continuous testing session on the same database instance, allowing issues like memory leaks to accumulate and eventually trigger crashes. In contrast, GQS restarts the database instance for each new graph to ensure a clean execution environment. This design choice enhances bug reproducibility and accelerates the detection of logic bugs, but may reduce the likelihood of revealing crash bugs that emerge only after prolonged execution. Given our focus on logic bug detection, we consider this trade-off reasonable.

6 Related Work

We have already discussed the state-of-the-art approaches for finding GDB logic bugs in Section 1. In this section, we focus on related work in relational database (RDB) testing and query synthesis.

RDB Testing. In recent years, a wide range of logic bug detectors have been developed for RDBs [3, 15, 20, 21, 46, 47, 49, 50, 54]. While RDBs and GDBs differ in their data models and query languages, many of their bug-detection techniques are grounded in similar principles—most notably, the use of randomized testing with randomly generated initial database states and queries. Moreover, test oracles such as differential testing and certain metamorphic testing strategies developed for RDBs can, in principle, be adapted to the GDB setting. GDBMeter illustrates this by adopting the metamorphic oracles from TLP [46], originally designed for SQL, which

partitions queries using three-valued logic. However, such adaptations may leave substantial parts of the graph-specific semantics untested. For example, key constructs in GDBs, such as paths, neighborhoods, and recursive traversals, do not map cleanly to the relational structures and operators in RDBs.

Similar to our approach, PQS [47], Pinolo [15], and TQS [54] test RDBs by synthesizing SQL queries. However, PQS and Pinolo cannot establish the ground truth and, instead, rely on subset relationships as test oracles—an approach that does *not* guarantee the correctness of individual queries. TQS targets logic bugs in join optimizations by synthesizing join queries based on normal forms. This is a strategy specific to the relational model that does not naturally extend to graphs. While our stepwise synthesis approach is tailored to the semantic characteristics of Cypher queries, many SQL queries, particularly those involving sequences of joins or recursion, exhibit conceptually similar chain-like structures. Hence, it should be possible to adapt our synthesis strategy to RDB testing as well.

Query Synthesis. SQL and Datalog query synthesis has been extensively studied over the past decades, with applications in database education, query optimization, and data migration. However, existing synthesizers [36, 52, 55, 56, 60] are *not* suitable for GDB testing. A key reason lies in their different application objectives. For instance, some synthesizers target educational use cases, prioritizing simplicity and explainability [56], whereas our goal is to generate complex queries capable of thoroughly exercising the behavior of GDBs.

Additionally, SQL and Datalog queries differ from graph queries like Cypher in both their structure and semantics, resulting in different synthesis tasks. Translating SQL or datalog queries to Cypher could enable the use of existing query synthesizers, originally developed for RDBs, in the context of GDB testing. Conversely, recent advances in transpiling Cypher to SQL suggest that our synthesized Cypher queries could, in principle, be adapted to RDB testing. However, state-of-the-art transpilers in both directions [25, 26, 38] either lacks soundness guarantees or may not yet support queries at the level of complexity generated by GQS.

7 Discussion and Concluding Remark

We have presented (i) GQS, the first automated testing approach for detecting logic bugs in GDBs with respect to an established ground truth, and (ii) an instantiation of GQS, incorporating the first Cypher query synthesizer tailored for GDBs. We have utilized GQS to uncover many previously unknown, long-latent bugs in production GDBs and demonstrated its superior effectiveness in bug detection compared to the state-of-the-art. Our approach is conceptually simple, facilitating its adoption in practice, yet highly effective, contributing to more reliable GDBs.

In the following, we discuss limitations and future work.

Limitations. While testing can find bugs in databases, it can generally not guarantee their absence [13, 30]. Our GQS approach, like other bug detectors, shares this inherent limitation. However, as demonstrated by our analysis, GQS is more effective than them in identifying logic bugs.

Due to the limitations of black-box testing, GQS may generate duplicate bug reports, as bugs with a common root cause can be triggered by seemingly distinct test cases. Currently, we rely on manual analysis to deduplicate these reports. Moreover, GQS does not support automated root cause analysis of detected bugs. However, compared to existing testers, it facilitates bug analysis by providing useful information, including the faulty database, the exact query execution, and the expected query result. In contrast, differential testers require developers to inspect each GDB under comparison to identify the faulty one, while metamorphic testers require manual examination of each query based on the metamorphic relations. Both automated bug deduplication and root cause analysis are orthogonal research areas in system testing and remain challenging [10, 57]. This represents a current research gap in GDB (and RDB) testing, which we hope to fill in the future.

In addition, GQS relies on the correctness of our implementation to accurately report logic bugs; otherwise, false alarms might be produced. However, formally verifying implementations at this scale is extremely challenging, as is recognized by the verification community [37, 51]. This is likely why no existing database testers have their implementations fully verified. Given the challenges of full formal verification, we instead sought to enhance confidence in the implementation through large-scale testing carried out with reasonable and practical effort.

GQS focuses on the common Cypher features shared across the tested GDBs to ensure compatible test queries, and it does not yet accommodate features specific to some GDBs, e.g., label expressions unique to Neo4j. Currently, GQS does not support subqueries or OLAP (Online Analytical Processing) queries. We plan to extend GQS with these additional features in future work.

Beyond Logic Bugs. Graph query generation is fundamental to GDB testing, not only for detecting logic bugs but also for assessing other properties. As already demonstrated by our experiments, the complex queries we synthesized are effective in triggering GDB crashes or unexpected exceptions (10 in total across four GDBs). Additionally, our query synthesizer can serve as a basis for examining GDB performance issues like inefficient query processing. The challenge lies in the current lack of ground truth on the expected execution time, where the cardinality estimation approach [45] designed for RDBs may provide a direction for future research.

Recent years have also seen advances in black-box checking of isolation levels [14, 17, 23, 28, 53, 58, 59], such as serializability, snapshot isolation, and weaker ones [5, 29]. These efforts primarily focus on RDBs and their concurrency control mechanisms for transactions (assuming the correctness for individual queries). Our work can be naturally extended to address this problem for GDBs, e.g., checking serializability claimed by Neo4j or snapshot isolation claimed by Memgraph. This can be done by wrapping generated Cypher queries as transactions and utilizing existing oracles like Adya's theory [2] or Biswas and Enea's axioms [7] to validate the fulfillment of different isolation guarantees.

Beyond Cypher. While Cypher is the *de facto* graph query language, other languages, such as Gremlin [48], are also widely used. To extensively test Gremlin-based GDBs, one can leverage the advanced *Cypher for Gremlin* compiler [41] to translate our synthesized Cypher queries into Gremlin queries. Using this approach, we tested JanusGraph [18] (v1.1.0) and uncovered two bugs within 24

hours. However, during this process, we encountered several limitations in the compiler's current support for Cypher features. For example, the compiler inaccurately translates `UNWIND` clauses and aggregation functions. To ensure correctness, we disabled these features. Improving the compiler in future work would likely enhance its effectiveness of bug detection for Gremlin-based GDBs.

As an alternative, our methodology can be applied to design a dedicated Gremlin synthesizer supporting its full range of features. This is feasible because Gremlin queries also employ a chain structure, comprising steps for individual graph traversal operations.

The GQL standard [1] has recently been published, marking a significant milestone in the database world. On one hand, our synthesizer will continue to be in action, as, e.g., Neo4j's compliance with GQL will not prevent any existing Cypher queries from functioning [32]. On the other hand, our approach and synthesizer lay the groundwork for effectively testing GQL-based databases in the future, as both languages naturally and deliberately converge, sharing a largely identical core syntax [32, 43].

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This research is supported by an ETH Zurich Career Seed Award.

References

[1] ISO/IEC 39075:2024. Accessed in July, 2024. Information technology — Database languages — GQL. https://www.iso.org/standard/76120.html.
[2] Atul Adya. 1999. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions.* Ph. D. Dissertation. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.
[3] Jinsheng Ba and Manuel Rigger. 2024. Keep It Simple: Testing Databases via Differential Query Plans. *Proc. ACM Manag. Data* 2, 3, Article 188 (2024).
[4] John W. Backus. 1959. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15-20 June 1959.* UNESCO (Paris), 125–131.
[5] Peter Bailis, Aaron Davidson, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* 7, 3 (2013), 181–192.
[6] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *ACM Comput. Surv.* 56, 2, Article 31 (sep 2023).
[7] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28.
[8] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs.* Morgan & Claypool Publishers.
[9] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. *CoRR* abs/2002.12543 (2020). arXiv:2002.12543 https://arxiv.org/abs/2002.12543
[10] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *PLDI '13*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 197–208.
[11] Cypher. Accessed in July, 2024. https://neo4j.com/docs/cypher-manual/current/introduction/.
[12] FalkorDB. Accessed in July, 2024. https://www.falkordb.com/.
[13] Shabnam Ghasemirad, Si Liu, Christoph Sprenger, Luca Multazzu, and David Basin. 2025. VerIso: Verifiable Isolation Guarantees for Database Transactions. *Proc. VLDB Endow.* 18, 5 (2025), 1362–1375.
[14] Long Gu, Si Liu, Tiancheng Xing, Hengfeng Wei, Yuxing Chen, and David Basin. 2024. IsoVista: Black-box Checking Database Isolation Guarantees. *Proc. VLDB Endow.* 17, 12 (2024).
[15] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *USENIX ATC '23.* USENIX Association, 345–358.
[16] 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 *ISSTA '23.* ACM, 163–174.
[17] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David A. Basin, Haixiang Li, and Anqun Pan. 2023. Efficient Black-box Checking of Snapshot Isolation in Databases. *Proc. VLDB Endow.* 16, 6 (2023), 1264–1276.
[18] JanusGraph. Accessed in April, 2025. https://janusgraph.org/.
[19] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *ICSE '24.* ACM, Article 46.
[20] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *OSDI '23.* 397–417.
[21] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *OSDI '24.* USENIX Association, 821–835.
[22] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *ISSTA '23.* ACM, 140–149.
[23] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280.
[24] Kùzu. Accessed in July, 2024. https://kuzudb.com/.
[25] Shunyang Li, Zhengyi Yang, Xianhang Zhang, Wenjie Zhang, and Xuemin Lin. 2021. SQL2Cypher: Automated Data and Query Migration from RDBMS to GDBMS. In *WISE 2021.* Springer-Verlag, 510–517.
[26] Jerry Liang. Accessed in April, 2025. openCypher Transpiler. https://github.com/microsoft/openCypherTranspiler.
[27] Cypher Parser Library. Accessed in July, 2024. libcypher-parser. https://github.com/cleishm/libcypher-parser
[28] Si Liu, Long Gu, Hengfeng Wei, and David Basin. 2024. Plume: Efficient and Complete Black-box Checking of Weak Isolation Levels. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024).
[29] Si Liu, Luca Multazzu, Hengfeng Wei, and David A. Basin. 2024. NOC-NOC: Towards Performance-optimal Distributed Transactions. *Proc. ACM Manag. Data* 2, 1, Article 9 (mar 2024).
[30] Si Liu, Peter Csaba Ölveczky, Min Zhang, Qi Wang, and José Meseguer. 2019. Automatic Analysis of Consistency Properties of Distributed Transaction Systems in Maude. In *TACAS 2019 (LNCS, Vol. 11428).* Springer, 40–57.
[31] Rupak Majumdar and Filip Niksic. 2017. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.* 2, POPL, Article 46 (Dec. 2017), 24 pages.
[32] Valerio Malenchino. Accessed in July, 2024. GQL is Here: Your Cypher Queries in a GQL World. https://neo4j.com/blog/cypher-gql-world/.
[33] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. In *ICSE '24.* ACM, Article 143, 12 pages.
[34] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107.
[35] Memgraph. Accessed in July, 2024. https://memgraph.com/.
[36] Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. 2021. GENSYNTH: Synthesizing Datalog Programs without Language Bias. *AAAI'21* 35, 7 (May 2021), 6444–6453.
[37] Peter Müller and Natarajan Shankar. 2021. The First Fifteen Years of the Verified Software Project. In *Theories of Programming: The Life and Works of Tony Hoare.* ACM Books, Vol. 39. ACM / Morgan & Claypool, 93–124.
[38] Neo4j. Accessed in April, 2025. SQL to Cypher translation. https://neo4j.com/docs/jdbc-manual/current/sql2cypher/.
[39] Neo4j. Accessed in July, 2024. https://neo4j.com/.
[40] openCypher. Accessed in July, 2024. https://opencypher.org/.
[41] openCypher. Accessed in July, 2024. Cypher for Gremlin. https://github.com/opencypher/cypher-for-gremlin.
[42] openCypher. Accessed in July, 2024. Cypher Query Language Reference, Version 9. https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf.
[43] Philip Rathle and Brad Bebee. Accessed in July, 2024. GQL: The ISO standard for graphs has arrived. https://aws.amazon.com/blogs/database/gql-the-iso-standard-for-graphs-has-arrived/.
[44] RedisGraph. Accessed in July, 2024. https://github.com/RedisGraph/RedisGraph.
[45] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *ESEC/FSE '20.* ACM, 1140–1152.
[46] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (nov 2020), 30 pages.
[47] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *OSDI '20.* USENIX Association, 667–682.
[48] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages (DBPL 2015).* ACM, 1–10.
[49] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query

Execution. In *ICSE '23*. IEEE Press, 2072–2084.
[50] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *Proc. VLDB Endow.* 17, 8 (may 2024), 1884–1897.
[51] Andrei Stefanescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Rosu. 2016. Semantics-based program verifiers for all languages. In *OOPSLA 2016*. ACM, 74–91.
[52] Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. 2021. PATSQL: Efficient Synthesis of SQL Queries from Example Tables with Quick Inference of Projected Columns. *Proc. VLDB Endow.* 14, 11 (2021), 1937–1949.
[53] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. COBRA: Making Transactional Key-Value Stores Verifiably Serializable *(OSDI '20)*. USENIX Association, Article 4.
[54] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proc. ACM Manag. Data* 1, 1, Article 55 (may 2023), 26 pages.
[55] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2021. Example-guided synthesis of relational queries. In *PLDI '21*. ACM, 1110–1125.
[56] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI '17*. ACM, 452–466.
[57] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *ISSTA '15*. ACM, 1–11.
[58] Hengfeng Wei, Jiang Xiao, Na Yang, Si Liu, Zijing Yin, Yuxing Chen, and Anqun Pan. 2025. Boosting End-to-End Database Isolation Checking via Mini-Transactions. In *ICDE 2025*. IEEE Computer Society, 3998–4010.
[59] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *EuroSys '23*. ACM, 654–671.
[60] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *ASE '13*. IEEE, 224–234.
[61] 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 *ISSTA '22*. ACM, 302–313.
[62] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. 2024. Testing Graph Database Systems via Graph-Aware Metamorphic Relations. *Proc. VLDB Endow.* 17, 4 (mar 2024), 836–848.