Dinkel: State-Aware and Granular Framework for Validating Graph Databases

Celine Wüst*
ETH Zürich
Switzerland
celine.wuest@inf.ethz.ch

Zuming Jiang*
ETH Zürich
Switzerland
The University of Hong Kong
Hong Kong SAR, China
jzuming@hku.hk

Zhendong Su
ETH Zürich
Switzerland
zhendong.su@inf.ethz.ch

ABSTRACT

Graph database management systems (GDBMSs) have been powering many data-driven applications. To ensure GDBMS reliability, several testing approaches have been proposed. However, they all suffer from two key limitations: (1) insufficient support for generating complex and valid queries to exercise deep GDBMS code, and (2) lack of general oracles to validate the execution correctness of arbitrary queries.

In this paper, we propose a novel and practical approach, Dinkel, for thoroughly testing GDBMSs. Our approach consists of two core techniques. First, to generate complex and valid queries, we model two kinds of graph state, *query context* and *graph schema*, to describe the Cypher variables and the manipulated graph labels and properties. We generate queries clause-by-clause, and modify the graph states on the fly to ensure each clause references the correct state information. Second, to generally validate query results, we introduce two fine-grained query transformations: clause-level and expression-level transformations. These transformations can operate on arbitrary queries while preserving their semantics. Dinkel validates GDBMSs by checking whether the transformed query produces the same results as the original. We evaluated Dinkel on three well-known GDBMSs. In total, we found 127 bugs, among which 113 were confirmed, 84 were fixed, and 33 were logic bugs. Compared to existing approaches, Dinkel can cover over 70% more code and find substantially more bugs within a 48-hour testing campaign. We expect Dinkel's powerful bug detection to lay a practical foundation for GDBMS testing.

PVLDB Reference Format:
Celine Wüst, Zuming Jiang, and Zhendong Su. Dinkel: State-Aware and Granular Framework for Validating Graph Databases. PVLDB, 19(6): 1198 - 1211, 2026.
doi:10.14778/3797919.3797928

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CelineWuest/dinkel.

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 19, No. 6 ISSN 2150-8097.
doi:10.14778/3797919.3797928

1 INTRODUCTION

Graph database management systems (GDBMSs) are crucial for modern interconnected, data-driven computer software [7, 10, 35, 36, 38, 52, 54]. Notably, 75% of the Fortune 100 and all of North America's top 20 banks have adopted the currently most popular GDBMS, Neo4j [12, 51]. While rapidly evolving and becoming more complex (*e.g.*, Neo4j has 1.2M LOC), GDBMSs inevitably suffer from bugs. One of the most critical types of bugs are logic bugs, which can silently cause GDBMSs to produce incorrect query results.

To improve GDBMS reliability, testing approaches [17, 19, 25, 59] have been proposed to find bugs by generating queries in Cypher, the most widely-adopted graph query language [39]. As large-scale, complex systems, GDBMSs are difficult to test effectively due to two key challenges: *query generation* and *test-oracle construction*. To cover a broad spectrum of functionalities and deep system logic, approaches need to generate complex queries involving various Cypher language features and complicated data dependencies. To identify bugs, especially logic bugs that cause GDBMSs to silently produce incorrect results for queries, testing approaches require test oracles that can be generally applied to validate arbitrary queries.

However, both of these challenges remain unsolved. All existing approaches focus on proposing new test oracles, but struggle to systematically generate complex and valid queries, leaving much GDBMS logic and code inadequately tested or completely unexercised. Although existing approaches focus on test oracles to identify logic bugs, none of the proposed oracles can generally be applied to validate arbitrary queries. As a result, many logic bugs in GDBMSs get missed, even though they can be triggered by existing queries. The following discusses the details of these two challenges.

Challenge 1: *Generating Complex Queries.* Many DBMS bugs can only be triggered by complex queries [21–23]. A complex Cypher query can involve various clause and complicated data dependencies. Generating such complex queries is challenging because the clauses invoked by Cypher queries can change the graph database states during their execution. Such state changes are visible in subsequent clauses. Moreover, the data and variables used by Cypher queries have different scopes, depending on the clauses they are involved in. Due to the lack of systematic modeling of these Cypher language features, existing approaches cannot effectively generate complex queries, and thus, many critical and deep logic of GDBMSs are not exercised by the testing campaigns of these approaches. Therefore, many bugs are missed by existing approaches.

Modeling Cypher language features is not trivial. Cypher is a declarative language that allows expressive data querying without describing specific control flows. However, different from typical

```
MERGE (x)<-[:A]-(x)<-[:A]-(x)<-[y:A]->(x)
DELETE y
CREATE (x)<-[:B]-()
DELETE y;
```

Figure 1: Assertion failure found by Dinkel in RedisGraph.

declarative query languages (*e.g.*, SQL), Cypher queries can make changes to the graph database state while executing Cypher clauses of the queries, whose effects are visible in subsequent clauses [11]. For example, a graph node created in an outer `CREATE` clause can be referenced by subsequent `DELETE` clauses. Such characteristics make it challenging to effectively test GDBMSs. On the one hand, neglecting these characteristics tends to make the generated queries simpler, resulting in the GDBMSs not having to handle complicated data dependencies and therefore not allowing the testing campaign to reach the deep logic of GDBMSs. On the other hand, failing to correctly handle such characteristics can make the generated Cypher queries invalid, resulting in many queries being discarded by GDBMSs in the early stages (*e.g.*, query parsing). For example, the generated queries may become invalid when referencing nonexistent or out-of-scope items.

Challenge 2: *General Test Oracle.* Existing approaches use query transformations to validate the correctness of query results [19, 25, 32, 59]. Given a generated Cypher query, they transform it into another query and check whether the results of the transformed query and the original query follow specified relationships. However, these approaches cannot be generally applied to arbitrary queries because they require their generated queries to follow specific query patterns. For example, GraphGenie [19] requires the predicate in `MATCH` clauses to satisfy the preconditions of their transformation rules (*e.g.*, one rule requires the predicates to involve a cyclic graph). Moreover, GraphGenie can transform and validate only queries that contain a single `MATCH` clause followed by a `RETURN`. GRev [32] integrates an abstract syntax graph (ASG) for interpreting the predicate in `MATCH` statements and can thus transform the predicate in more diverse ways and with fewer restrictions. However, the predicate modeling in GRev can only be applied to `MATCH` statements, while queries involving other common clauses like `MERGE`, `UNWIND`, and `FOREACH` remain unsupported and thus untested.

Such inapplicabilities arise from the coarse-grained transformation used by existing approaches. Specifically, their transformations are at the query level. When transforming a query to another query, these approaches need to consider the whole query semantics and choose transformations according to these parsed semantics. They cannot handle semantics that do not fit their transformations (*e.g.*, both GraphGenie and GRev do not support `FOREACH` clauses). Such coarse-grained methods make existing approaches inapplicable to various queries that do not match specific query patterns. Therefore, many logic bugs are missed. For example, existing approaches miss the bug shown in Figure 2 because the original query is too complex and does not follow any specified patterns.

In this paper, we propose Dinkel, which is designed to lay a practical foundation for testing GDBMSs. Dinkel integrates two technical solutions to address the two challenges discussed above.

Solution 1: *State-Aware Query Generation.* Dinkel models graph states and state changes caused by Cypher queries. It abstracts the states into two categories, *query context* and *graph schemas*.

```
// Original: 2 rows
RETURN 0 AS n1
UNION CALL {
  FOREACH (n2 IN [] |
    FOREACH (n3 IN [] |
      MERGE (:A {u:0})))
} MATCH (y) RETURN y AS n1
UNION CREATE ()
RETURN 0 AS n1
```

```
// Transformed: 1 row
FOREACH (n0 IN [] |
  FOREACH (x IN [] | SET x = {}))
RETURN 0 AS n1
UNION CALL {
  FOREACH (n2 IN [] |
    FOREACH (n3 IN [] |
      MERGE (:A {u:0})))
} MATCH (y) RETURN y AS n1
UNION CREATE ()
RETURN 0 AS n1
```

Figure 2: Queries exposing a logic bug in Neo4j.

Query context contains information related to temporary variables declared in the queries (*e.g.*, the type and scope of each variable), while graph schemas maintain the current graph information, including the graph labels and properties. As query context and graph schemas may change at different Cypher clauses, we must update these abstractions while generating Cypher queries. To this end, we propose *state-aware query generation*. Instead of determining query skeletons for later expressions complementarily, our approach incrementally constructs clauses for the generated queries. When constructing a new Cypher clause, our approach references only the accessible elements according to the current query context and graph schema. After the clause is completed, the approach updates the corresponding state information. In this on-the-fly way, our approach can accurately maintain the dynamically evolving graph state and thus generate queries involving complicated data dependencies and state changes.

Solution 2: *Fine-Grained Query Transformation.* Our idea to validate Cypher queries is inspired by EET [23], which is designed for SQL queries in relational DBMSs (RDBMSs). EET proposes to validate SQL queries by transforming them at the level of expressions. In this paper, we further generalize this method to *fine-grained query transformation*, without limiting the transformation units to just expressions. Specifically, instead of analyzing the semantics of a whole query, our approach transforms queries by their fine-grained units. In the context of Cypher queries, the fine-grained units can not only be expressions, but also clauses, which can flexibly change graph states. By operating on these low-level units, we focus on the fine-grained semantics of queries, without the need to analyze the overall query-level semantics. For example, we can transform the first `RETURN` clause of the original query in Figure 2 by inserting an additional `FOREACH` clause, which is dead code and has no effects because the array to be iterated over is empty. Therefore, the clauses before and after transformations should be semantically equivalent, and the transformed query should produce the same results as the original one. However, their results differ, indicating a logic bug triggered. In this process, Dinkel only needs to ensure the semantic equivalence between clauses, without understanding the semantics of the whole query.

Regarding the transformation for clauses, Dinkel manipulates target clauses to affect the graph state in ways that do not interfere with the semantics of the following clauses (*e.g.*, creating and immediately deleting a node), or introduce dead code before target clauses. Regarding the transformation for expressions, given an arbitrary expression within clauses, Dinkel can construct a semantically equivalent expression by leveraging logical (*e.g.*, De

Table 1: Grammar of Cypher query language

query	::=	*clause* [*query*]
clause	::=	*reading_clause* | *writing_clause* | *reading/writing_clause* | *projecting_clause* | ...
reading_clause	::=	['OPTIONAL'] 'MATCH' *pattern* ['WHERE' *expression*]
pattern	::=	*node* [*relationship pattern*]
node	::=	'(' *label** *properties?* ')'
relationship	::=	'<-[' *label properties?* ']-' | '-[' *label properties?* ']->'
label	::=	':' *identifier*
properties	::=	'{' *identifier* : *expression* [, *identifier* : *expression*] '}'
expression	::=	*identifier* | *constant* | *operation* | *function* | ...
writing_clause	::=	*create_clause* | *delete_clause* | *set_clause* | *remove_clause* | *foreach_clause* | ...
reading/writing_clause	::=	*merge_clause* | *call_clause* | ...
projecting_clause	::=	*return_clause* | *with_clause* | *unwind_clause* | ...

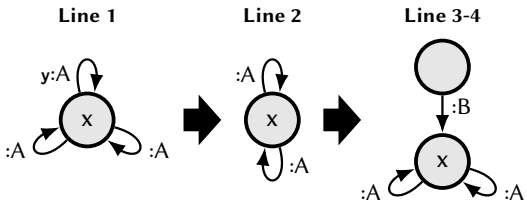

Figure 3: State changes of the query in Figure 1.

Morgan's laws) and algebraic equivalences (*e.g.*, `x ≡ x + 0`). Dinkel also uses features specific to the Cypher language for transforming expressions (*e.g.*, `x ≡ CASE WHEN true THEN x END`).

We implemented Dinkel as a fully automatic GDBMS testing framework. In the evaluation on Neo4j [37], RedisGraph [40], and Memgraph [34], Dinkel efficiently generated complex Cypher queries and kept a high validity rate (89.02%). Dinkel found 127 unique new bugs, among which 33 are logic bugs. 113 bugs have been confirmed, and 84 fixed. Many bugs are long-latent and missed by existing approaches. Compared to existing approaches, Dinkel can cover over 70% more code and find more bugs within the 48-hour testing campaign. These results demonstrate that Dinkel significantly outperforms existing approaches.

In summary, we make the following contributions:

- ***Novel approach:*** We tackle two fundamental problems of GDBMS testing via two novel solutions: (1) *state-aware query generation*, which abstracts the Cypher state information to facilitate query generation; and (2) *fine-grained query transformation*, which can be generally applied to validate arbitrary Cypher queries by manipulating their clauses and expressions.
- ***Practical realization:*** We realize a fully automatic testing framework, Dinkel, to find bugs in GDBMSs by generating and validating complex Cypher queries.
- ***Promising results:*** We evaluated Dinkel on three popular open-source GDBMSs, namely Neo4j, RedisGraph and Memgraph. Dinkel found 127 unique and new bugs, among which 33 are logic bugs. So far, 113 unique bugs have been confirmed, and 84 fixed. These results demonstrate the effectiveness of Dinkel.

2 BACKGROUND

Graph Database Management Systems. GDBMSs utilize graph models to represent data. The most widely used graph model is the *labeled property graph model* [19], which stores interconnected data using nodes connected via relationships (*i.e.*, directed edges between nodes) [39]. Nodes and relationships, generally referred to as *graph entities*, can be associated with labels and properties. Labels are used to group and classify elements, whereas properties are made up of key-value pairs, providing attribute information.

Cypher Language. Cypher is the most widely adopted query language for property graph databases [39]. It is a declarative language. The general way to specify data in Cypher is to concretize the *graph patterns*, which follow the format `(n)-[r]->(m)` and can be used to reference graph entities satisfying specified conditions. For example, the query in Figure 1 concretizes a graph pattern following the `CREATE` to specify the graph entities to be created.

Unlike SQL, Cypher does not differentiate between data declaration (DDL), manipulation (DML), or query (DQL) languages. Instead, Cypher can create, read, and modify data in a single query, thereby allowing queries of procedural nature and non-trivial state manipulation. Cypher queries operate on graphs via clauses. Table 1 shows the context-free grammar of Cypher. A Cypher query consists of a sequence of clauses. Each clause can be either a reading, writing, reading/writing, projecting, or other clauses (*e.g.*, system configuration clause) [11]. Reading clauses (*i.e.*, `MATCH`) fetch information without modifying the graph entities. Writing clauses (*e.g.*, `CREATE`, `DELETE`) modify the data stored in GDBMSs by changing the nodes or relationships in the graph. Reading/writing clauses (*e.g.*, `MERGE`) can both read and write data. Projecting clauses (*e.g.*, `WITH`, `RETURN`) define expressions to be referenced in the subsequent clauses or the result set. GDBMSs sequentially process the clauses in a query, and the graph state may change after each clause is processed.

Variables can be referenced by subsequent clauses within their scopes. The variable scopes are affected by specific clauses. These clauses are `WITH`, `CALL`, `UNION`, and `FOREACH`. For example, in the transformed query shown in Figure 2, the second `FOREACH` can reference the `n0` defined in the outer `FOREACH`, but the subsequent `RETURN` cannot, as it is outside the body of the first `FOREACH`. Similarly, the last `CREATE` clause cannot reference the node `y` from the previous `MATCH (y)`, as they are separated by a `UNION` clause. Clauses following a `CALL` can only reference variables explicitly returned from it. For example, the `FOREACH` at the end of Figure 6 can reference `n3` but not `n1`. A `WITH` keeps only explicitly mentioned variables alive, while also possibly declaring new ones. The `WITH` at the start of the `CALL` in Figure 6 declares a new variable `n1` with value `[]` and kills all other variables. Exceptionally, the `WITH *` keeps all variables alive.

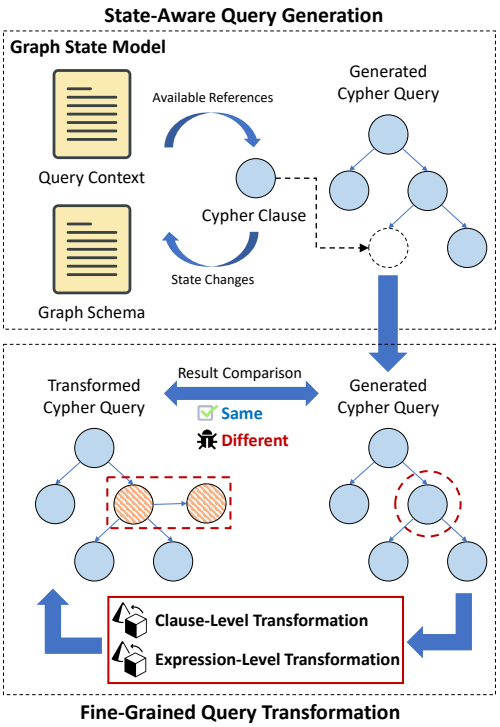

Figure 4: Approach overview.

Illustrative Example. Figure 1 shows a query triggering an assertion failure in RedisGraph. Figure 3 shows the corresponding graph state changes during execution of each its clause. The first clause, `MERGE`, creates graph entities following the specified graph pattern as no graph entity matches the pattern. A node `x` and three relationships with labels `A` are created. Each relationship is from `x` to `x`. The subsequent `DELETE` checks whether the graph entity `y` exists and deletes the existing ones, ultimately deleting one relationship. The `CREATE` creates a new node connected to `x` with a new relationship `B`. The last `DELETE` tries to delete `y`, but as `y` has already been deleted, this `DELETE` should do nothing. However, RedisGraph recognized `y` as non-deleted and performed a deletion on a nonexistent relationship, which triggered an assertion failure. Such a mistake is caused by the entity ID reuse mechanisms of RedisGraph, which reassigns the ID of the deleted `y` to the new relationship created by the `CREATE`. When referencing `y` in the last clause, RedisGraph mistakenly considers `y` to still exist because its entity ID is being used.

Cypher queries with multiple clauses changing graph states are commonly used in the real world due to the complex nature of data relationships within graph databases. However, it is extremely challenging to automatically generate and validate such queries, because (1) the generation needs to be aware of the graph state changes caused by each clause (*e.g.*, `MERGE` and `DELETE`) and properly reference the intermediate data (*e.g.*, variables `x` and `y`); and (2) the validation needs to construct general test oracles in the case of the complicated semantics the query can contain.

3 APPROACH

We propose Dinkel, a GDBMS testing framework addressing the above challenges. Figure 4 shows its overview.

Algorithm 1: State-Aware Query Generation

```
Output: query
Function GenQuery():
    query ← EmptyQuery();
    qc ← {}; // query context
    gs ← {}; // graph schema
    do
        clause, qc, gs ← GenClause(qc, gs, ANY);
        AppendClause(query, clause);
    while rand() < P and Length(query) < L_max;
    return query;
Function GenClause(qc, gs, type):
    if type = ANY then
        type = RandClauseType();
    // initialize the clause accordingly
    clause, qc, gs ← RandInitClause(qc, gs, type);
    foreach subclause, subclause_type in clause do
        subclause, qc, gs ← GenClause(qc, gs,
        subclause_type);
    // clean out-of-scope query context
    CleanQC(qc);
    return clause, qc, gs;
```

3.1 State-Aware Query Generation

3.1.1 Graph State Modeling. Cypher query execution can be affected by two kinds of graph states:

Query Context. It contains temporary variables declared in the query. These variables have types and scopes, affecting only the query state where they are declared. They can either be concrete values (*e.g.*, integer 0) or aliased to specific nodes or relationships (*e.g.*, node `x` and relationship `y` in Figure 1). Such variables can be referenced only after they are defined, within their scope. We refer to these variables, their types, and their scope information as *query context*. Query context may change at different Cypher clauses. Specifically, query context includes new information when a new variable is declared by a clause, and excludes outdated information when the clause is out of the scope of existing variables.

Graph Schema. It manages the schema of the stored graph data. It includes graph labels and properties (*e.g.*, label `A` and `B` for relationships in Figure 1). We refer to such state information as *graph schema*. Graph schemas can be changed by specific clauses. For example, `CREATE` clauses can create nodes or relationships with new labels and properties, while `REMOVE` clauses can remove existing labels or properties. Different from query context, whose effects are limited by their scope, the operations (*e.g.*, `CREATE` clause that declares new labels) on graph schema affect the database permanently. In addition, graph schema can be referenced even though the labels and properties are nonexistent. This design improves query flexibility as users can write valid queries without concerning themselves with the current graph schema.

3.1.2 Query Generation. Our generation is based on a key insight that clauses are the basic units for operating on graphs, and both query context and graph schema are updated only when clauses

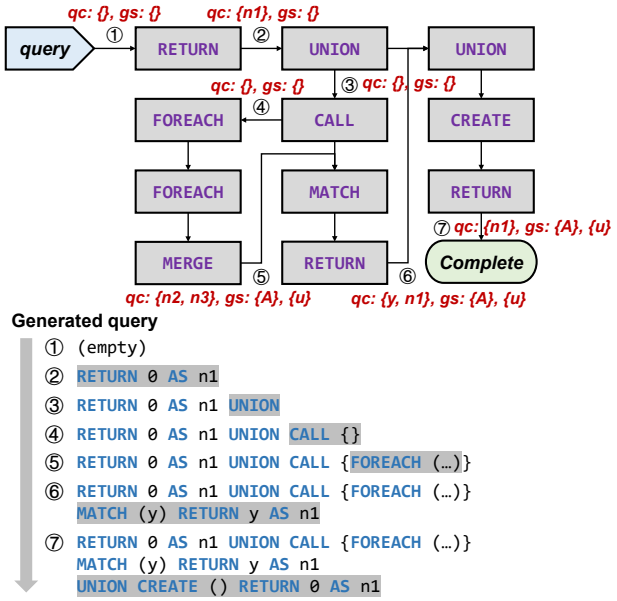

Figure 5: Generation for the original query in Figure 2.

are invoked or exit specific clause context. Based on this insight, our approach track the state changes inside Cypher queries by analyzing the possible impact caused by specific clauses in queries.

Algorithm. Algorithm 1 shows the procedure of our state-aware query generation. Our approach does not need any input. Initially, it constructs an empty query that contains no clause, and initializes query context *qc* and graph schema *gs* to empty sets (line 2-4). Then, the approach incrementally appends the query with a newly generated clause (line 5-8). Each time a new clause is generated, the query context and graph schema may be updated (line 6). Our approach decides whether to append the query with more clauses based on a certain probability P. If the length of the query reaches the limit L_{max}, the approach stops appending to the query (line 8). In the end, the generated query is returned (line 9).

To generate a clause, our approach references the query context and graph schema. If the clause type is not specified, the approach randomly chooses a clause type (line 11-12). Based on thi information, our approach randomly initializes a clause (line 13). Initialization for different clauses can vary (*e.g.*, the procedures for `CALL` and `CREATE` are different), but it generally includes the procedure of determining the components needed by the clause, generating expressions for these components, and analyzing the impacts of the generated clause. If the generated clause contains subclauses, the approach will recursively call `GenClause()` to generate subclauses with specified types (line 14-15). After the clause is generated, the approach cleans up the query context if necessary (line 16), such as removing out-of-scope variables at the end of a `CALL` clause. In the end, the generated clause, the updated query context, and the graph schema are returned (line 17).

3.1.3 Illustrative Example. Figure 5 shows how we generate the original query in Figure 2. Initially, the query is empty, and both query context and graph schema contain nothing (step ①). The approach randomly determines that the first clause is a `RETURN` with an integer `0` as the returned value `n1`. Then, the approach updates the query context by adding `n1` with its type and scope (step ②). The second clause is a `UNION`. After constructing the `UNION`, the approach cleans the query context as the clause after the `UNION` cannot reference the variables defined before the `UNION` (step ③). Subsequently, the approach randomly decides to generate a `CALL` clause with a subclause (step ④). The subclause is recursively generated as a `FOREACH`, defining `n2` in its loop condition. The `FOREACH` also has a subclause, which is another `FOREACH` defining `n3` and containing a `MERGE`. Our approach records the query context (*i.e.*, `n2`, `n3` defined by the two `FOREACH`) and graph schema (*i.e.*, label `A` and property `u` referenced by the `MERGE`) updated in this recursive generation process (step ⑤). After completing these `FOREACH` clauses, the approach removes `n2` and `n3` from query context as they are scoped to only the `FOREACH` clauses. Finishing the `CALL`, our approach generates a `MATCH` defining `y` and a `RETURN` referencing `y` as `n1`, and updates the query context by adding `y` and `n1` (step ⑥). Finally, our approach randomly generates a `UNION` followed by a `CREATE` and a `RETURN`. The `UNION` clears the query context, after which `RETURN` adds a new variable `n1` (step ⑦).

Table 2: Clause-level transformation rules. (1) T.S.: transformation strategies including Dead Code (DC), Unused Context (UC), Redundant Write (RW), and Inconsequential Supersession (IS); (2) `<literal>`: any atomic expression; (3) `<mirror(x)>`: a mirrored version of `x` (*e.g.*, `a->b` ⇒ `b<-a`); (4) `<clauses>`: a random number of random clauses; (5) `<new pattern>`: a graph pattern that involves at least one previously unused label or property

Type	Original	T.S.	Transformed
Read-Only	`C`	DC	`CALL {MATCH x WHERE false <clauses>} C`
		DC	`CALL {MATCH <new pattern> <clauses>} C`
		IS	`OPTIONAL MATCH <new pattern> C`
		UC	`CALL {WITH * RETURN x AS i} C`
		UC	`UNWIND <literal> AS i C`
	`WITH x`	UC	`WITH x, y AS i`
	`MATCH x`	IS	`MATCH x WHERE true`
		IS	`MATCH <mirror(x)>`
Write/ Project	`C`	DC	`FOREACH (i IN [] |<clauses>)C`
		RW	`DELETE <deleted var> C`
		RW	`CREATE x DELETE x C`
		RW	`SET x += {} C`
	`REMOVE x`	RW	`REMOVE x, x`
	`DELETE x`	RW	`DELETE x, x`
		IS	`DETACH DELETE x`
Any	`C`	IS	`WITH * C`

3.2 Fine-Grained Query Transformation

To enable general query validation, we construct test oracles based on the fine-grained units of queries: clauses and expressions.

3.2.1 Clause-Level Transformations. These transformations manipulate queries by either adding new clauses, or switching out clauses with equivalent counterparts. Table 2 lists the transformation rules. We design different rules for reading clauses and writing/projecting clauses because the Cypher specification limits the kinds of clauses that can be used before specific clauses. The transformations consider state information to avoid semantic invalidity (*e.g.*, repeatedly defining the same variables).

Table 3: Expression-level transformation rules. (1) T.S.: transformation strategies including Mathematical Identity (MI) and Cypher Feature (CF); (2) `P'`: randomly generated boolean expression; (3) `E'`: randomly generated expression

Type	Initial	T.S.	Transformed
Bool	`P`	MI	`NOT (NOT P)`
		MI	`P OR false`
		MI	`P AND true`
	`P OR Q`	MI	`NOT ((NOT P)AND (NOT Q))`
	`P AND Q`	MI	`NOT ((NOT P)OR (NOT Q))`
	`true`	MI	`true OR P'`
		MI	`P' OR (NOT P')OR (P' IS NULL)`
	`false`	MI	`false AND P'`
		MI	`P' AND (NOT P')AND (P' IS NOT NULL)`
	`NULL`	MI	`NULL {XOR|=|<>|<|>|>=|<=} P'`
Num	`x`	MI	`-(-x)`
		MI	`x {+|-} 0`
		MI	`x {*|/|^} 1`
	`0`	MI	`0 * x`
	`NULL`	MI	`NULL {+|-|*|/} x`
Str	`S`	MI	`S + ""`
List	`L`	MI	`L + []`
Any	`E IS NULL`	MI	`NOT (E IS NOT NULL)`
	`E`	CF	`CASE WHEN true THEN E ELSE E' END`
		CF	`CASE WHEN false THEN E' ELSE E END`

We introduce four strategies to achieve semantic-preserving transformation for clauses: *Dead Code (DC)*, *Unused Context (UC)*, *Redundant Write (RW)*, and *Inconsequential Supersession (IS)*. *Dead Code* expands the target clause by introducing additional clauses that, due to Cypher's way of handling queries, will not be executed. We implement three rules for this strategy. Two rules transform read clauses by adding a `CALL` with a `MATCH`. The `MATCH` is designed to return no rows, leading to no rows for the clauses following the `MATCH` to ingest. As a result, the subsequent clauses in the `CALL` will not be executed. One rule transforms write clauses by inserting a `FOREACH` before them. The `FOREACH` iterates the elements in a given array and invokes clauses to operate on the iterated elements. Because the array is empty, the invoked clauses in the `FOREACH` will not be executed. For example, in Figure 2, our approach transforms the first `RETURN` of the original query by adding a dead `FOREACH`, which preserves the semantics of the `RETURN` and thus should not affect the final results. However, the results of the transformed query change, indicating that a bug was triggered.

Unused Context expands a clause by involving unused query context. Specifically, it defines new variables that are not referenced by subsequent parts of the query. *Redundant Write* is straightforward. It expands a clause by deleting a previously deleted node, removing the same property multiple times, or creating and immediately deleting a subgraph. Lastly, *Inconsequential Supersession* expands a clause by adding inconsequential operations (*e.g.*, adding `WHERE true` to a `MATCH` clause) or modifying the formats of the clauses (*e.g.*, reversing the graph pattern of a `MATCH` clause).

Clause-level transformations have the ability to influence the query context through manipulating variables, as well as changing the database state by modifying graph elements or affecting control flow. By doing so, Dinkel can effectively force the GDBMSs to execute system logic that was not covered by the original queries, and thus validate the query results by checking whether the different executed logic produces the same results.

3.2.2 Expression-Level Transformations. These transformations operate on expressions within Cypher clauses. Although they do not modify graph states, they still force the GDBMS to evaluate more complex expressions. Table 3 lists corresponding transformations.

These transformations involve two strategies: *Mathematical Identity (MI)* and *Cypher Feature (CF)*. *Mathematical Identity* leverages mathematical laws (*e.g.*, De Morgan's laws for boolean con- and disjunctions, and double negations) to transform the expressions while preserving their semantics. *Cypher Feature* leverages the features of the Cypher language. For example, Dinkel can transform an arbitrary expression into a `CASE` expression, where the branch condition is `true`, and the corresponding value of the true branch is the original expression. When transforming expressions, Dinkel sometimes needs to involve new components in the transformed expressions (*e.g.*, the `E'` in the transformed `CASE` expressions). In this case, Dinkel constructs these components by referencing the available variables recorded in the corresponding query context.

4 IMPLEMENTATION

We implement our approach as a fully automatic GDBMS testing framework, Dinkel. The overall codebase consists of 10k lines of Go code. The following discusses important implementation details.

Query Generation. Dinkel supports all non-administrative clauses in the tested GDBMSs. For example, in Neo4j, 21 out of the total 25 clauses are non-administrative and thereby supported. Administrative clauses, such as `SHOW FUNCTIONS` and `USE`, are not supported because they are not related to query processing logic. For example, `SHOW FUNCTIONS` lists available functions and cannot be combined with other clauses. `USE` is used to switch the manipulated databases, which is not necessary as all databases are empty on initialization and they function in the same way.

As shown in Table 4, Dinkel supports more Cypher clauses than existing approaches [17, 19, 25, 32, 59]. None of the existing approaches support `FOREACH` clauses, because this clause complicates control flow and can significantly modify graph states during query execution. `UNION`, `EXISTS`, and `COUNT` are not supported either, as these clauses invoke subqueries that can inherit and change the graph states of the main query. Lacking a systematic model for handling graph state changes, existing approaches are limited in generating complex queries using these clauses. Moreover, existing approaches support validation for only queries following specific patterns, while many clauses complicate the semantics of Cypher queries, making them unable to be validated by existing approaches.

When generating a query, Dinkel choose a unique random seed to determine the construction process of clauses and expressions. To later reproduce the same sequence of clauses and expressions, the seed gets recorded and stored alongside the automatically generated bug report. As the seed is represented by a 64-bit integer, it is very unlikely for seeds to be reused.

Query Reduction. No existing tool is available for reducing bug-triggering Cypher queries. The queries may contain dozens of

Table 4: Cypher clauses supported by existing approaches

Clause	GDsmith	GDBMeter	GraphGenie	GAMERA	Dinkel
`MATCH`	●	●	●	●	●
`CREATE`	●	●	●	●	●
`MERGE`	○	○	○	○	●
`DELETE`	○	●	○	○	●
`REMOVE`	○	●	○	○	●
`SET`	○	●	○	○	●
`UNWIND`	●	○	●	●	●
`WITH`	●	○	●	●	●
`RETURN`	●	●	●	●	●
`CALL`	○	○	●	○	●
`FOREACH`	○	○	○	○	●
`UNION`	○	○	○	○	●
`EXISTS`	○	○	○	○	●
`COUNT`	○	○	○	○	●

clauses, with only few of them being related to the root causes of a bug. Redundant clauses not only confuse developers, but also seriously impact the bug localization [4, 15, 50]. For example, commit bisection [15] cannot test commits older than the supporting of redundant clauses, even though these commits introduced the bugs. To address this problem, we implemented Dinkel with automatic query reduction. It reduces queries clause by clause. For each clause, Dinkel tries to delete it. If the query without the clause still triggers the bug, the clause will be permanently removed. Otherwise, the deleted clause will be recovered, and Dinkel goes on to try to replace the clause with an alternative clause if possible. For example, Dinkel can try to replace the `CALL` clause in Figure 2 with its subclause `CREATE`. If some clauses are successfully deleted or replaced in one try, Dinkel will restart the reduction process for the reduced queries. The process stops when no clause in the query can be further reduced. This final query then acts as a minimal test case for the bug it detects, allowing developers to refer to it for addressing the underlying issue and verifying their fixes.

5 EVALUATION

Our evaluation aims to answer the following questions:

Q1 Can Dinkel find real bugs in widely used and extensively tested GDBMSs? (Section 5.2)

Q2 How complex and valid are the queries generated by Dinkel? (Section 5.3)

Q3 How do different techniques contribute to the effectiveness of Dinkel? (Section 5.4)

Q4 Can Dinkel outperform state-of-the-art GDBMS testing approaches? (Section 5.5)

5.1 Experimental Setup

We evaluated Dinkel on Neo4j [37], RedisGraph [40], and Memgraph [34], which are popular and extensively tested by existing approaches [17, 19, 25, 32, 59]. We evaluate Dinkel on the latest GDBMS versions. During testing, if the tested GDBMSs are updated (*e.g.*, a new version is released), we set up new Dinkel instances to test the updated versions. Specifically, we test Neo4j from version 5.6.0, RedisGraph from 2.12.0, and Memgraph from 2.7.0. RedisGraph is no longer maintained after 2.12.10, and thus we move to test its successor, FalkorDB [13], from its first release version 4.0.0. We count the bugs of RedisGraph and FalkorDB together as they share the majority of their code base. To show its effectiveness for black-box testing GDBMSs, we evaluate Dinkel on the Enterprise version of Neo4j from 5.6.0 onward. Each time we implement new features on Dinkel, we stop and restart the testing. Overall, the testing campaign intermittently lasted 9 months. The evaluation was performed on a machine running Ubuntu 20.04, with a 64-core AMD EPYC 7742 processor running at 2.25GHz and 256GB of RAM.

Table 5: Status of the bugs found by Dinkel

GDBMS	Reported	Confirmed	Fixed
Neo4j	51	51	49
RedisGraph	62	48	29
Memgraph	14	14	6
Total	127	113	84

5.2 Bug Detection

Table 5 shows the status of the bugs found by Dinkel. In total, Dinkel found 127 unique bugs, including 51 bugs in Neo4j, 62 in RedisGraph, and 14 in Memgraph. Among these bugs, 113 are confirmed, and 84 are fixed. None of the 127 bugs are duplicates. All the tested GDBMSs have been extensively tested in both industry and academia. The significant number of new bugs demonstrates the powerful bug-finding ability of Dinkel.

Bug Classification. We classify the bugs found by Dinkel into three categories according to their manifestation:

- *Logic bugs.* The tested GDBMSs produce incorrect results for specific queries. These bugs were exposed because they lead to discrepancies between the results of the original queries and the transformed queries generated by Dinkel.
- *Internal errors.* The tested GDBMSs unexpectedly throw exceptions or errors when processing syntactically and semantically valid queries. The error messages can indicate the inconsistency of the internal execution status.
- *Crashes.* The queries cause GDBMSs to crash due to assertion failures or memory corruption.

Table 6 shows the classification results. Among the 127 bugs, 33 are logic bugs, 56 cause internal errors, and 38 result in GDBMS crashes. Note that Neo4j is implemented in Java and has exception handling for unexpected memory errors. Thus, Neo4j does not crash, but rather responds with exception information when triggering memory corruption. Therefore, Dinkel found no crashes in Neo4j, but 38 internal errors. Among the 38 crashes found in RedisGraph and Memgraph, 29 are caused by memory corruptions, and 9 are caused by assertion failures. These results demonstrate that Dinkel can comprehensively test GDBMSs by finding various bugs.

Figure 6 shows a bug that triggers a Neo4j internal error. This query constructs an empty array `[]` and invokes a subquery using a `CALL`. The subquery also constructs an empty array. It then invokes two `UNWIND` clauses, which iterate over each element in the operated array. For each iterated element, the `UNWIND` executes the subsequent clause under the context of this element. For example, the array `[x]` used by the second `UNWIND` references the variable `x`, which is `0` when the first `UNWIND` iterates over the item `0` in the array `[0]`. After

Table 6: Classifying the found bugs

GDBMS	Logic Bug	Internal Error	Crash
Neo4j	13	38	0
RedisGraph	18	10	34
Memgraph	2	8	4
Total	33	56	38

```
WITH [] AS n0 ORDER BY null
CALL {
  WITH [] AS n1 ORDER BY null
  UNWIND [0] AS x
  UNWIND [x] AS n2
  RETURN n2 AS n3
} FOREACH (n4 IN null | MERGE ({key:n3}))
```

Figure 6: A query triggering a Neo4j internal error—Neo4j-Error: ExecutionFailed (arraycopy: last destination index 6 out of bounds for object array[5]).

```
MATCH (x)
CALL {
    WITH x
    MATCH ({n0:x.n1})
    MATCH (:A)
    WITH * ORDER BY x
    RETURN 0 AS n2
} RETURN 0
```

Figure 7: A query triggering a segmentation fault in RedisGraph from accessing an invalid memory location.

the `CALL`, the query utilizes `FOREACH`, whose execution depends on the query context of the `CALL`. To optimize query execution, Neo4j tries to flatten the `FOREACH` loop. However, such optimization does not work well when the loop involves update operations (*i.e.*, `MERGE`) under complicated contexts (*i.e.*, the contexts produced by `CALL`). The improper optimization corrupts the internal data structures of Neo4j, *Eagers*, which are the production of another optimization, *Eagerness analysis*. In the end, an internal error is triggered when Neo4j tries to access the corrupted Eagers. To fix this bug, Neo4j developers modify both the Eagerness analysis and the flattening strategy for `FOREACH` clauses to ensure they work consistently.

Figure 7 shows the query triggering a crash in RedisGraph. At the start of the query, any node is matched and assigned to a variable with the `MATCH (x)`. As the graph is empty, this `MATCH` should match no node. Next, a `CALL` subquery is initiated, taking over the matched nodes from the outside context using `WITH x`. After that, any node is matched whose property `n0` corresponds to the property `n1` of the matched nodes in `x`. Since no node was matched, and therefore nothing is bound to `x`. Therefore, `x` is assigned `NULL`. Subsequently, a `MATCH` matches any node with a label of `A`, of which none exist. A `WITH` clause references `x` in its `ORDER BY` expression. The `CALL` subquery exits by returning the value `0`, bound to the variable `n2`. Finally, the value `0` is returned. The segmentation fault is triggered when RedisGraph clears the contexts of the `CALL`. The expression `x.n1` in the second `MATCH` has an invalid base `x`, which is `NULL` and causes a null-pointer dereference. The developers of RedisGraph created a bounty for this bug, rewarding the person who fixes it with $100 for their efforts.

Table 7: GDBMS components affected by bugs. The number in the parentheses is for the logic bugs

GDBMS	Parser	Planner	Executor
Neo4j	18 (0)	18 (5)	10 (3)
RedisGraph	15 (1)	15 (3)	19 (3)
Total	23 (1)	23 (8)	29 (6)

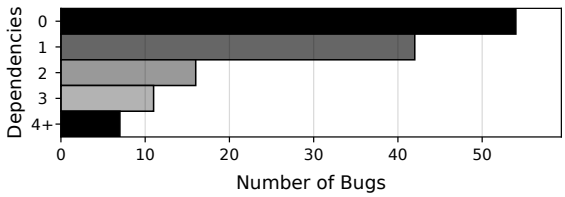

Figure 8: Data dependencies of bug-triggering queries.

This bug was fixed by adding checks to make sure that properties being accessed have a valid base. In general, developers fixing crashes and other availability-impacting bugs found by Dinkel helped in reducing potential downtime for GDBMS customers.

Affected GDBMS Components. We investigated 59 of the 84 fixed bugs, where we were able to analyze the fix patches to identify the GDBMS components affected by the bugs accurately. The 19 fix patches in the Enterprise version of Neo4j are confidential and thus not included. The 6 fixed bugs in Memgraph are also not included because Memgraph developers mix the fixes into their regular development commits, where we cannot accurately analyze the affected components. Table 7 shows the results. Among the 59 fixed bugs, 29 affect the parsers of GDBMSs, 25 affect the planners, and 20 affect the executor. Interestingly, 29 bugs (5 logic bugs) affect two components, and 12 bugs (1 logic bug) affect all the components. For example, the bug shown in Figure 11 affects both the planner and executor of Neo4j. In total, 40 bugs (8 logic bugs) affect either the planner, the executor, or both. The 6 logic bugs affecting the executor also affect the planner. These results demonstrate that (1) Dinkel can find bugs in various GDBMS components; and (2) Dinkel can find bugs in the deep logic of GDBMSs, considering 67.8% (40/59) of fixed bugs are related to the planners or executors.

Data Dependencies for Triggering Bugs. To convey the data dependencies required to trigger bugs, we analyze dependencies in the 127 bug-triggering queries. The results are shown in Figure 8. Over half of the bugs (58%) require at least one data dependency. Specifically, 42 bug-triggering queries contain one data dependency, 16 contain two, 11 contain three, and 7 contain four or more. For example, in Figure 1, the query contains 8 data dependencies, as illustrated in Figure 9. Among the 8 dependencies, 6 are involved in query context (*i.e.*, the variables `x` and `y` assigned to a node and a relationship), and 2 are involved in graph schema (*i.e.*, the label `A`). These results demonstrate that some GDBMS bugs can be triggered only when the queries contain multiple data dependencies, and Dinkel can effectively find these bugs.

Size and clause number of Bug-Triggering Queries. Figure 10 shows the amount of clauses used by the 127 bug-triggering queries,

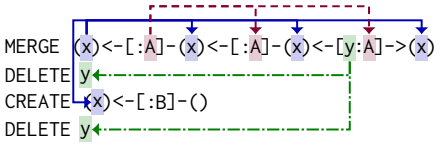

Figure 9: Data dependencies within the query in Figure 1.

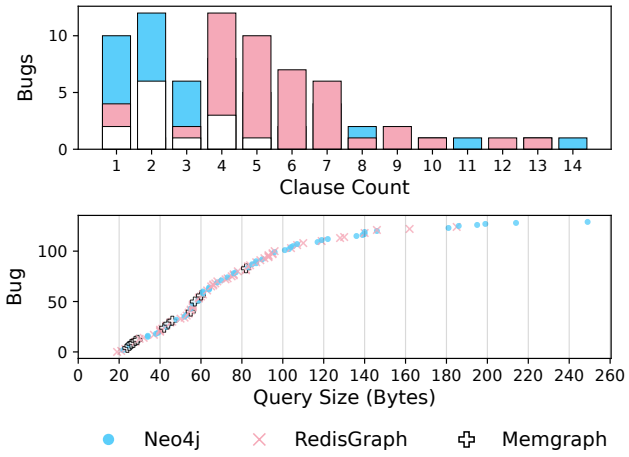

Figure 10: Clause counts and sizes of bug-triggering queries.

as well as their size. All these queries have been reduced using the methods mentioned in Section 4. 118 bugs can be triggered using 7 clauses or fewer, and 112 bugs can be triggered by queries whose size is less than 120 bytes. Few bugs require at least 8 clauses to be triggered. The bug that needed the most amount of clauses to be triggered is shown in Figure 2, using 14 clauses in the transformed query. As query size increases from 0 to 120 bytes, the number of triggered bugs increases almost linearly. The bug-triggering query with the largest size is 249 bytes, which triggers an internal error in Neo4j and was fixed by their developers.

Bug Importance. Neo4j, the GDBMS we focus on, provides Community and Enterprise versions. The bugs in the Enterprise version are critical because this version is commonly deployed on commercial applications. Among the 51 Neo4j bugs Dinkel found, 32 bugs can be triggered in both Enterprise and Community versions, and 19 bugs can be triggered in only the Enterprise version. RedisGraph and Memgraph did not provide severity information about the reported bugs, but we noticed that most bugs were fixed, indicating that the bugs are non-trivial. Some developers express their appreciation for our effort in finding bugs in their GDBMSs. Most notably, Neo4j provides very positive feedback, personally thanking us via e-mail and sending a gift-box.

5.3 Query Generation

We ran Dinkel on each tested GDBMS for 48 hours and collected all generated queries (transformed queries are not included) to understand query complexity and validity. Table 8 shows the results.

Query Complexity. On average, Dinkel generates 655k queries for each GDBMS, with 22.62M clauses constructed. The average number of clauses of each query is 34.54, and the number of data dependencies is 26.36. Because of the large number of clauses and dependencies, the queries generated by Dinkel are typically large. The average query size is 1153.3 bytes. These results demonstrate that Dinkel can effectively generate complex queries, which are large and contain multiple clauses with complicated data dependencies. The generated queries are more complex than the bug-triggering queries discussed in Section 5.2, because all the bug-triggering queries are reduced, while the generated queries inevitably contain many redundant parts [22, 31, 41].

Table 8: Queries generated by Dinkel within 48 hours. Gen.: the number of generated queries; Valid: the number of valid queries; Clause, Dep., and Size: the average of clauses, data dependencies, and bytes over all queries, respectively

GDBMS	Gen.	Valid	Clause	Dep.	Size
Neo4j	318k	89.90%	40.97	26.77	1394.69
RedisGraph	1'078k	93.90%	32.71	30.59	1071.58
Memgraph	570k	83.27%	29.95	21.71	993.63
Average	655k	89.02%	34.54	26.36	1153.3

Query Validity. Among the 655k queries generated by Dinkel, 583k are valid. The percentage of valid queries is 89.02%. The result demonstrates that Dinkel can keep a high validity rate even when generating complex queries. We investigated the invalid queries generated by Dinkel and found that they were mainly caused by illegal arithmetic operations. Some arithmetic operations require the operands to satisfy some constraints (*e.g.*, the divisor in a division must be non-zero). However, Dinkel cannot track the value of each operator in queries because the value may depend on complex expressions, whose results are difficult to calculate (*e.g.*, hash functions). Queries failing to satisfy the constraints of operations cause semantic errors, such as divisions by zero.

Throughput. On average, one Dinkel instance generates 3.79 tests (each test consists of one original query and one transformed query) per second. Specifically, at each second, one Dinkel instance completes 1.84 tests for Neo4j, 6.24 for RedisGraph, and 3.30 for Memgraph. To understand the throughput bottleneck, we investigated the time used by Neo4j in query generation, transformation, and execution. We found that for Neo4j, 74.75% of CPU time was used to execute the generated queries, which are complex and can be time-consuming. This result indicates that the test throughput is dominated by the performance of the tested GDBMSs. We believe the current throughput is practical considering (1) GDBMS testing typically lasts for several months [19, 25], and thus a sufficient number of test cases can be executed; and (2) setting up multiple Dinkel instances can significantly improve the test efficiency.

5.4 Importance of Key Techniques

To show the importance of our techniques, we analyze how they contribute to the 127 bugs found by Dinkel. The analysis consists of two parts. The first focuses on the query context and graph schema in query generation, while the second assesses the transformation strategies (4 clause-level ones and 2 expression-level ones) used in test-oracle construction.

We analyzed the 127 bug-triggering queries and checked whether their generations depend on specific graph states. Specifically, we

Table 9: Bug-triggering queries related to QC: only query context; GS: only graph schema; Both: both graph states. The number in the parentheses is for the logic bugs

Version	Found	QC	GS	Both
Neo4j	51 (13)	23 (3)	3 (0)	5 (2)
RedisGraph	62 (18)	34 (7)	1 (1)	4 (0)
Memgraph	14 (2)	5 (0)	1 (0)	0 (0)
Total	127 (33)	72 (10)	5 (1)	9 (2)

Table 10: Transformation strategies used in finding logic bugs, including Dead Code (DC), Unused Context (UC), Redundant Write (RW), Inconsequential Supersession (IS), Mathematical Identity (MI), and Cypher Feature (CF)

Type	Strategy	Neo4j	RedisGraph	Memgraph	Total
Clause	DC	3	4	0	7
	UC	2	3	0	5
	RW	1	3	1	5
	IS	2	6	0	8
	Total	8	16	1	25
Expr	MI	1	1	1	3
	CF	4	1	0	5
	Total	5	2	1	8
Total		13	18	2	33

extracted the data dependencies contained by each query and check whether at least one dependency is related to query context or graph schema. Table 9 shows the results. Among the 127 queries, 72 depend on only query context, 5 depends on only graph schema, and 9 depend on both. According to these results, without query context, 81 bugs (12 logic bugs) cannot be found. This demonstrates the importance of query contexts, which enable Dinkel to generate queries referencing specific nodes, relationships, or expressions constructed in internal clauses. Without graph schema, 14 bugs (3 logic bugs) will be missed. These bug-triggering queries require Dinkel to properly reference the latest graph labels or properties.

We investigated the 33 logic bugs found by Dinkel, and checked whether their bug-triggering queries use specific transformation strategies. Table 10 shows the results. We find that each logic bug was triggered by using just one transformation rule in Table 2 and Table 3. Specifically, among the 33 logic bugs, 25 are exposed by clause-level transformations, and 8 by expression-level transformations. The number of bugs found by each transformation strategy is close to the average (*i.e.*, 5.5). Inconsequential Supersession finds the most bugs with 8, and Mathematical Identity finds the least with 3 bugs. These results indicate that all strategies are effective in identifying logic bugs in GDBMSs. Moreover, the 25 bugs found by clause-level transformations demonstrate the effectiveness of our fine-grained transformation over EET [23], which supports only expression-level transformation.

The following discusses two representative bug examples.

Example 1: Figure 11 shows a Neo4j bug that causes an incorrect query result. The original query references both query contexts and graph schema. Specifically, the first `CREATE` clause creates two

```
// Original: 1
CREATE ({x:0})<-[y:A]-()
CREATE ({x:endNode(y).x})
RETURN COUNT {({x:0})} AS n
```

```
// Transformed: 2
CREATE ({x:0})<-[y:A]-()
CREATE (z) DELETE z
CREATE ({x:endNode(y).x})
RETURN COUNT {({x:0})} AS n
```

Figure 11: Clause-level transformation reveals a bug.

```
// Original: NULL
RETURN NULL AS x
```

```
// Transformed: FALSE
RETURN NULL XOR EXISTS{()} AS x
```

Figure 12: Expression-level transformation reveals a bug.

nodes with a relationship `y`, labeled `A`. One node has a property `x` with value `0`. The second `CREATE` clause creates a new node with a property `x` whose value is equal to the property `x` of the end node of `y` (*i.e.*, `0`). The `RETURN` clause returns the number of nodes whose property `x` is `0`. The expected result should be 2. However, Neo4j produces 1, because of a logic bug triggered when Neo4j processes two consecutive `CREATE`s that have data dependencies. Dinkel finds this bug using a clause-level transformation, Redundant Write. It inserts a `CREATE` clause and `DELETE` clause between the two `CREATE`s. Such a transformation preserves the query semantics but makes the two `CREATE`s not consecutive anymore, and thus Neo4j produces a correct result for the transformed query. The developers fixed this bug by significantly reconstructing the source code related to `CREATE` clauses. In the end, 17 source files were modified.

Example 2: Figure 12 shows a logic bug in Neo4j found by expression-level transformation, Mathematical Identity. It is based on a rule that the result of `NULL XOR a` is `NULL`, no matter what `a` is. Dinkel transforms the `NULL` into `NULL XOR EXISTS{()}`, expecting that the result should remain `NULL`. However, Neo4j produces `FALSE`. The root cause of this bug is related to an operator in the Cypher planner of Neo4j, LetSelectOrSemiApply. It is invoked only when the transformed query is processed. Such an operator fails to work correctly because it does not consider `NULL` expressions in its arguments. To fix this bug, Neo4j developers refined this operator by adding additional logic to check and handle `NULL` in its arguments.

5.5 Comparison

Bug Latency Study. To demonstrate that Dinkel can find bugs missed by existing approaches, we investigate the latencies of bugs found by Dinkel. For each bug, we check whether its bug-inducing commit was created before the years when existing approaches were published. If Dinkel finds some long-latent bugs, we can conclude that Dinkel can find bugs missed by existing approaches. This comparison is reasonable and objective because: (1) none of the bugs found by Dinkel are marked as duplicated by developers, meaning that no approach found these bugs until Dinkel found them; and (2) all existing approaches have extensively tested Neo4j and RedisGraph [17, 19, 25, 32, 59], meaning that in these two GDBMSs, no approach found the long-latent bugs found by Dinkel during their evaluation. Some existing approaches do not test Memgraph, but we still include it to show the effectiveness of Dinkel.

GDsmith [17], GDBMeter [25], GraphGenie [19], and GAMERA [59] were published in 2023, and GRev [32] was published in 2024. Therefore, for each bug, we checked whether its bug-inducing commit was created before 2024 (*i.e.*, in 2023 or earlier) and 2023. As

Table 11: Latency of the bugs found by Dinkel. The number in the parentheses is for the logic bugs

DBMS	Found	Bug-involved year		Longest latency
		< 2024	< 2023	
Neo4j	51 (13)	40 (9)	27 (7)	2016
RedisGraph	62 (18)	52 (15)	20 (3)	2020
Memgraph	14 (2)	9 (2)	4 (0)	2021
Total	127 (33)	101 (26)	51 (10)	2016

Table 12: The clauses used by bug-triggering queries. The number in the parentheses is for the logic bugs. N/A indicates that the GDBMS does not support the clause

Clause	Neo4j	RedisGraph	Memgraph	Total
`MATCH`	20 (7)	28 (6)	5 (1)	53 (14)
`CREATE`	17 (9)	26 (11)	2 (1)	55 (21)
`MERGE`	12 (4)	31 (12)	5 (1)	48 (17)
`DELETE`	4 (4)	16 (6)	1 (1)	21 (11)
`REMOVE`	0	1 (0)	0	1 (0)
`SET`	4 (2)	3 (1)	0	7 (3)
`UNWIND`	9 (1)	11 (6)	1 (0)	21 (7)
`WITH`	16 (3)	23 (8)	0	39 (11)
`RETURN`	40 (8)	39 (13)	11 (1)	90 (22)
`CALL`	15 (3)	22 (8)	4 (0)	41 (11)
`FOREACH`	8 (5)	6 (3)	0	14 (8)
`UNION`	10 (3)	7 (3)	1 (0)	18 (6)
`EXISTS`	10 (3)	N/A	N/A	10 (3)
`COUNT`	10 (4)	N/A	N/A	10 (4)

shown in Table 11, among the 113 bugs in Neo4j and RedisGraph, 47 already existed before 2023, among which 10 are logic bugs. It indicates that all existing approaches failed to find these 47 bugs in their extensive evaluation. In addition, 45 bugs (24 logic bugs) were introduced to Neo4j and RedisGraph between 2023 and 2024, while GRev, which was proposed after 2024, cannot find these bugs. In addition, GDsmith and GRev support Memgraph, but neither can find the 4 long-latent bugs induced before 2023. These results indicate that existing approaches indeed miss many long-latent bugs, while Dinkel can effectively find them.

Clause Analysis. We analyze the clauses used in the 127 bug-triggering queries and show the results in Table 12. The results indicate that the bugs found by Dinkel are related to various Cypher clauses. Combining Table 4 and Table 12, we can demonstrate that many bugs found by Dinkel cannot be found by existing approaches, because they do not even support the clauses used in some bug-triggering queries (*e.g.*, 14 bugs (8 logic bugs) related to `FOREACH` clauses). As discussed in Section 4, existing approaches cannot support these clauses because they lack a systematic model for query generation and a general test oracle for query validation.

Empirical Comparison. To empirically demonstrate that Dinkel outperforms the state-of-the-art on code coverage and bug detection, we further evaluated Dinkel and existing approaches on Neo4j and RedisGraph, which are the only two GDBMSs supported by all these approaches. Each evaluation persisted for 48 hours and was repeated 5 times. Table 13 shows the comparison results.

Table 13: Results of covered lines (average ± standard deviation) and found bugs of existing approaches over 5 runs

	Neo4j		RedisGraph	
	Line (Avg ± SD)	Bug	Line (Avg ± SD)	Bug
Dinkel	750k ± 5.2k	19	20k ± 0.1k	26
GDBMeter	283k ± 0.2k	1	19k ± 0.3k	1
GRev	597k ± 4.0k	1	14k ± 0.1k	1
GAMERA	283k ± 0.1k	1	3k ± 0.1k	0
GDsmith	515k ± 8.6k	0	15k ± 0.1k	0
GraphGenie	506k ± 2.0k	0	3k ± 0.1k	0

Code Coverage. As shown in Table 13, on average, Dinkel covers 71% and 85% more code than existing approaches in Neo4j and RedisGraph, respectively. Among the tools, excluding Dinkel, GRev covers the most code (*i.e.*, 597k lines) in Neo4j. Compared to GRev, Dinkel covers 25% more code. Benefiting from the powerful query generation, Dinkel can cover much deeper logic of GDBMSs that is related to processing advanced Cypher features and complicated data dependencies. In contrast, simple queries generated by existing tools rarely trigger such logic. In RedisGraph, excluding Dinkel, GDBMeter covers the most code (*i.e.*, 19k lines). Compared to it, Dinkel covers 5% more code. The coverage improvement is less significant than in Neo4j, because RedisGraph supports fewer Cypher features and cannot handle some complicated Cypher semantics. For example, RedisGraph does not support processing the subqueries in `FOREACH` and `CALL` clauses. As a result, the space for Dinkel to improve the code coverage in RedisGraph is limited.

Found Bugs. Table 13 shows the number of bugs found by each approach over 5 runs. Dinkel finds the most bugs in both Neo4j and RedisGraph, where logic bugs comprise 4 of the bugs found in Neo4j and 7 in RedisGraph. In Neo4j, GDBMeter, GRev, and GAMERA found the same 1 bug. In RedisGraph, GRev and GDBMeter found the same 1 bug. Both of these bugs were also identified by Dinkel. For the 43 bugs missed by existing approaches, their bug-triggering queries contain either complicated data dependencies or Cypher features that are only supported by Dinkel (as discussed in Section 4). The bugs shown in Figure 11 and Figure 12 are examples of these 43 bugs. These results demonstrate that Dinkel can find more bugs than existing approaches by generating and validating complex queries.

6 LIMITATIONS AND FUTURE WORK

Concurrency Issues. Dinkel is not designed to detect concurrency issues (*e.g.*, data races). During testing, Dinkel sets up only a single client that connects to a database and executes queries. Therefore, concurrency issues can be found only if they occur on single connections. For example, a crash found by Dinkel is caused by a data race in RedisGraph with a single connection. Moreover, Dinkel cannot effectively find bugs related to the executions of concurrent transactions, which require multiple connections to be set up. One interesting future work is to facilitate Dinkel with concurrency testing [6, 18, 53] to find more various bugs.

Faults Handling. Another class of bugs out of scope of Dinkel are bugs in fault handling of GDBMSs. As large-scale systems, GDBMSs need to ensure the consistency when faults (*e.g.*, crashes caused

by power loss) happen. Dinkel cannot effectively test the implementation related to fault handling because faults rarely occur. One possible future work is to combine Dinkel with fault injection techniques [1, 20, 60]. By injecting faults to GDBMSs (*e.g.*, intentionally shutting down the database), the approach can force the systems to execute the fault-handling code and thus have chances to detect bugs in the corresponding executions.

Adaption to New GDBMS features. GDBMSs are rapidly developing, together with many new features implemented and existing features modified. Dedicated testing for these features is motivated when GDBMSs evolve. Current Dinkel randomly generate Cypher queries and lacks support for dedicated testing. In the future, we plan to parameterize the query generation of Dinkel, so that developers can adjust corresponding parameters to make generation focus on specific features. Furthermore, directed fuzzing techniques [5, 27, 45] can be integrated into Dinkel to guide testing to cover specific code snippets related to new GDBMS features.

7 RELATED WORK

GDBMS Testing. GDBMS testing is an emerging research field, where several approaches [17, 19, 25, 32, 57, 59] have been proposed. All these approaches construct test oracles to identify logic bugs. Grand [57] targets GDBMSs using the Gremlin query language and detects logic bugs using differential testing [33]. Several approaches [17, 19, 25, 32, 59] support GDBMSs using Cypher. GDBMeter [25] decomposes the predicate of a query and checks if the queries with the decomposed predicates produce consistent results with the original query. GraphGenie [19] modifies the graph patterns used in a query and checks if the results of the original query and the query with modified graph patterns satisfy the expected relationship (*i.e.*, equivalence, subset, or superset) . GAMERA [59] extracts metamorphic relations [9] from the manipulated graph data and checks whether the outputs of generated queries satisfy the extracted relations.

Dinkel is different from all existing approaches in two key aspects. First, Dinkel significantly improves query generation, the core foundation of GDBMS testing. Second, while existing approaches support correctness testing for only queries following specific patterns, Dinkel generally tackles the problems of test-oracle construction, enabling validation for arbitrary queries.

RDBMS Testing. Compared to GDBMS testing, testing relational database management systems (RDBMSs) is more mature, where both query generation [14, 21, 28, 48, 58] and test-oracle construction [16, 23, 42–44, 46, 49] are well-researched. SQLsmith [48] embeds the SQL grammar [47] and can generate complex SQL statements. DynSQL [21] incrementally generates SQL statements for a query by querying the latest DBMS schema, which enables DynSQL to decouple the generation of each statement and helps it generate queries containing multiple complex statements. To find logic bugs in RDBMSs, PQS [44] synthesizes customized queries that fetch specific rows of tables. If the tested RDBMS fails to fetch the rows, a logic bug is identified. Pinolo [16] modifies the predicate of a query and checks whether the modified query produces the subset/superset of the results of the original query. EET [23] transforms the expressions of queries in a semantic-preserving manner and checks whether the queries with the transformed expressions produce the same results as the original queries. EET identifies a bug if their results are not the same. Due to the separation of querying and data manipulation in the SQL language, EET cannot perform clause-level transformations that Dinkel supports for Cypher queries.

Unlike these approaches for SQL queries, Dinkel is designed for validating Cypher queries. Leveraging Cypher features, Dinkel integrates novel techniques for both query generation and test-oracle construction. Compared to SQL, Cypher queries can be much more flexible, being able to both read and write within a single query, while also including more complex control flows. As a result, the state changes within Cypher are more intricate, making techniques for SQL generation infeasible to generate complex Cypher queries. Dinkel tackles this problems by modeling the possible state changes involved by Cypher clauses, and construct Cypher queries based on the tracked graph states. Additionally, the test oracle of Dinkel is inspired by EET [23], but it go more systematically. Dinkel is not constrained to only transforming expressions like in EET. Instead, it can also manipulate the database state within a query by operating the clauses with stored data in a semantics-preserving manner.

State-Aware Fuzzing. Fuzzing is a promising technique for finding bugs in software [2, 8, 20, 24, 29, 55]. Some approaches [3, 21, 26, 30, 56] have been proposed to find bugs more efficiently in state-sensitive systems. RESTler [3] analyzes API specifications of the tested cloud services and generates request sequences that follow the inferred producer-consumer dependencies. RESTler also collects the response observed during prior requests to guide subsequent request generation. LOKI [30] proposed to test blockchain consensus protocols. It builds a state model to dynamically track the state transition of each node in the blockchain systems and accordingly generates inputs with proper targets, types, and contents. StateFuzz [56] is designed to test Linux drivers. It utilizes static analysis to recognize critical variables that affect control flows or memory accesses, and represents program states using these variables. StateFuzz prioritizes test cases that trigger new states. For effectively testing USB gadget stacks, FuzzUSB [26] extracts the internal state machines from USB gadget drivers via static analysis and symbolic execution, before using this state information as fuzzing feedback for guiding test-case generation.

Different from these approaches, Dinkel models GDBMS-related state information for allowing query generation to introduce complex data dependencies, all while retaining query validity.

8 CONCLUSION

We have presented a novel and practical framework, Dinkel, for validating GDBMSs. We model the graph state as query context and graph schema, and propose state-aware query generation to generate complex and valid Cypher queries for testing the deep logic of GDBMSs. Moreover, we propose two fine-grained query transformations: clause-level transformations and expression-level transformations, which can operate on arbitrary Cypher queries to validate their correctness. In our evaluation, Dinkel found 127 bugs in three well-known GDBMSs, among which 33 are logic bugs. Considering its significant advancement in bug finding, we believe Dinkel can lay a practical foundation for GDBMS testing, facilitating and inspiring follow-up research on GDBMS reliability.

REFERENCES

[1] A framework for distributed systems verification, with fault injection. 2024. https://github.com/jepsen-io/jepsen.

[2] American Fuzzy Lop. 2021. https://github.com/google/AFL.

[3] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. 748–758.

[4] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. 2018. Orca: Differential bug localization in Large-Scale services. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 493–509.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 International Conference on Computer and Communications Security (CCS)*. 2329–2344.

[6] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *Proceedings of the 29th USENIX Security Symposium*. 2325–2342.

[7] Jingji Chen and Xuehai Qian. 2023. DecoMine: A Compilation-Based Graph Pattern Mining System with Pattern Decomposition. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 47–61.

[8] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 2018 Symposium on Security and Privacy (S&P)*. 711–725.

[9] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases*. Technical Report. HKUST-CS98-01. Department of Computer Science, HKUST.

[10] Creating a Modern Solution to Finding What's Lost with Neo4j. 2024. https://neo4j.com/case-studies/notlost.

[11] Cypher Query Language Reference. 2022. https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf.

[12] DB-Engines Ranking. 2024. https://db-engines.com/en/ranking.

[13] FalkorDB. A super fast Graph Database uses GraphBLAS under the hood for its sparse adjacency matrix graph representation. 2024. https://github.com/FalkorDB/FalkorDB/.

[14] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of the 37th International Conference on Automated Software Engineering (ASE)*. 1–12.

[15] Git-bisect: Use binary search to find the commit that introduced a bug. 2024. https://git-scm.com/docs/git-bisect.

[16] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *Proceedings of the 2023 USENIX Annual Technical Conference (ATC)*. 345–358.

[17] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of the 2023 International Symposium on Software Testing and Analysis (ISSTA)*. 163–174.

[18] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P)*. 754–768.

[19] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland Hock Chuan Yap, Zhenkai Liang, and Manuel Rigger. 2023. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. 531–542.

[20] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2020. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *Proceedings of the 29th USENIX Security Symposium*. 2595–2612.

[21] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *Proceedings of the 32nd USENIX Security Symposium*. 4949–4965.

[22] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 397–417.

[23] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 821–835.

[24] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*.

[25] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *Proceedings of the 2023 International Symposium on Software Testing and Analysis (ISSTA)*. 140–149.

[26] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. 2022. FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks. In *Proceedings of the 2022 International Symposium on Security and Privacy (S&P)*. 2212–2229.

[27] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, 3559–3576.

[28] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-Oriented DBMS Fuzzing. In *Proceedings of the 2023 International Conference on Data Engineering (ICDE)*. 668–681.

[29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Security Symposium*. 1949–1966.

[30] Fuchen Ma, Yuanliang Chen, Meng Ren, Yuanhang Zhou, Yu Jiang, Ting Chen, Huizhong Li, and Jiaguang Sun. 2023. LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols. In *Proceedings of the 30th Network and Distributed System Security Symposium (NDSS)*.

[31] David Maciver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP)*. 13:1–13:27.

[32] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. 1–12.

[33] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[34] Memgraph. Open-source graph database, tuned for dynamic analytics environments. Easy to adopt, scale and own. 2024. https://github.com/memgraph/memgraph.

[35] NBC News Analyzes Hundreds of Thousands of Russian Troll Tweets Using Neo4j. 2021. https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-case-study-NBC-News-EN-US.pdf.

[36] Neo4j and Generative AI 2023. https://neo4j.com/generativeai/.

[37] Neo4j. Graphs for Everyone. 2024. https://github.com/neo4j/neo4j.

[38] Novartis Captures the Latest Biological Knowledge for Drug Discovery. 2021. https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-case-study-Novartis-EN-US.pdf.

[39] openCypher. 2024. https://opencypher.org/.

[40] RedisGraph. A graph database as a Redis module. 2023. https://github.com/RedisGraph/RedisGraph.

[41] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 2012 International Conference on Programming Language Design and Implementation (PLDI)*. 335–346.

[42] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-optimizing Reference Engine Construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESE/FSE)*. 1140–1152.

[43] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. In *Proceedings of the 2020 International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 1–30.

[44] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 667–682.

[45] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. MC^2: Rigorous and Efficient Directed Greybox Fuzzing. In *Proceedings of the 2022 International Conference on Computer and Communications Security (CCS)*. 2595–2609.

[46] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. 2072–2084.

[47] SQL standard. 1992. https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[48] SQLsmith: A random SQL query generator. 2023. https://github.com/anse1/sqlsmith.

[49] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. In *Proceedings of the 2023 International Conference on Management of Data (SIGMOD)*. 1–26.

[50] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*. 262–273.

[51] Who Uses Neo4j. 2024. https://neo4j.com/who-uses-neo4j.

[52] Chengshuo Xu, Keval Vora, and Rajiv Gupta. 2019. Pnp: Pruning and prediction for point-to-point iterative graph analytics. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 587–600.

[53] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data race fuzzing for kernel file systems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 1643–1660.

[54] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2023. Glign: Taming misaligned graph traversals in concurrent graph processing. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 78–92.

[55] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium*. 745–761.

[56] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing. In *Proceedings of the 31st USENIX Security Symposium*. 3273–3289.

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

[58] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 International Conference on Computer and Communications Security (CCS)*. 955–970.

[59] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. 2023. Testing Graph Database Systems via Graph-Aware Metamorphic Relations. In *Proceedings of the 50th International Conference on Very Large Databases (VLDB)*. 836–848.

[60] Yonghao Zou, Jia-Ju Bai, Zu-Ming Jiang, Ming Zhao, and Diyu Zhou. 2025. Blackbox Fuzzing of Distributed Systems with Multi-Dimensional Inputs and Symmetry-Based Feedback Pruning. In *Proceedings of the 32nd Network and Distributed System Security Symposium (NDSS)*.