2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) | 979-8-3503-2263-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE-COMPANION58688.2023.00041

Randomized Differential Testing of RDF Stores

Rui Yang*†, Yingying Zheng*†, Lei Tang*†, Wensheng Dou*†‡§, Wei Wang*†‡§, Jun Wei*†‡§

*State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences

†University of Chinese Academy of Sciences

‡University of Chinese Academy of Sciences Nanjing College

§Nanjing Institute of Software Technology

{yangrui22, zhengyingying14, tanglei20, wsdou, wangwei, wj}@otcaix.iscas.ac.cn

***Abstract*—As a special kind of graph database systems, RDF stores have been widely used in many applications, e.g., knowledge graphs and semantic web. RDF stores utilize SPARQL as their standardized query language to store and retrieve RDF graphs. Incorrect implementations of RDF stores can introduce logic bugs that cause RDF stores to return incorrect query results. These logic bugs can lead to severe consequences and are likely to go unnoticed by developers. However, no available tools can detect logic bugs in RDF stores.**

In this paper, we propose *RD*^{2}, a Randomized Differential testing approach of RDF stores, to reveal discrepancies among RDF stores, which indicate potential logic bugs in RDF stores. The core idea of RD^2 is to build an equivalent RDF graph for multiple RDF stores, and verify whether they can return the same query result for a given SPARQL query. Guided by the SPARQL syntax and the generated RDF graph, we automatically generate syntactically valid SPARQL queries, which can return non-empty query results with high probability. We further unify the formats of SPARQL query results from different RDF stores and find discrepancies among them. We evaluate RD^2 on three popular and widely-used RDF stores. In total, we have detected 5 logic bugs in them. A video demonstration of RD^2 is available at https://youtu.be/da7XlsdbRR4.

***Index Terms*—RDF store, differential testing, SPARQL**

I. Introduction

The Resource Description Framework (RDF) [1] graph model has been regarded as a W3C standard for exchanging graph data. We refer to the graph database systems (GDBs) that are built on the RDF graph model as RDF stores [2]. The representative RDF stores include MarkLogic [3], Apache Jena [4], GraphDB [5], RDF4j [6], etc. They utilize SPARQL [7] as their standardized query language, and play a significant role in knowledge graphs [8], [9] and semantic web [10].

Similar to other GDBs that are built on the labeled property graph model [11]–[13] and relational database systems (RDBMSs) [14]–[18], incorrect implementations of RDF stores can introduce logic bugs that result in an incorrect query result for a given SPARQL query, e.g., omitting a record. Fig. 1 illustrates a real-world logic bug found in MarkLogic. In this example, we first write an RDF triple (Line 2) into MarkLogic, and then retrieve it by a FILTER expression $80596426678 * 1719307142$, which should be evaluated into $true$ (Line 5−7). We expect the RDF triple can be returned (Line 9). However, MarkLogic mistakenly returns an empty result (Line 8) because the expression is evaluated into $false$ due to decimal overflow. But, Apache Jena and RDF4j can correctly return the RDF triple.

```
1  <!-- RDF graph data -->
2  <http://JohnSmith> <http://ages> 12.
3
4  <!-- SPARQL query -->
5  SELECT *
6  WHERE{ ?s ?p ?o .
7    FILTER ( 80596426678*1719307142 )}
8  -- {} ✘
9  -- {<http://JohnSmith> <http://ages> 12} ✔
```

Fig. 1. A logic bug in MarkLogic [3].

Logic bugs in RDF stores can easily go unnoticed by developers. Existing GDB testing approaches, e.g., Grand [11], GDsmith [12] and GDBMeter [13], detect logic bugs in Gremlin-based or Cypher-based GDBs, which adopt the labeled property graph model. Existing RDBMS testing approaches [14]–[20] detect bugs in RDBMSs. However, RDF stores adopt totally different graph models and query syntaxes. Existing approaches cannot be directly applied on RDF stores.

In this paper, we propose a randomized differential testing technique *RD*^{2}, to reveal discrepancies among RDF stores. These discrepancies usually indicate logic bugs in RDF stores. We first randomly build an equivalent RDF graph for multiple RDF stores and write it into multiple target RDF stores. We then generate random SPARQL queries. Finally, we compare the query results returned by these RDF stores to check whether discrepancies exist. To effectively reveal discrepancies among RDF stores, we address two specific challenges. (1) Randomly generated SPARQL queries can return empty query results with high possibility. This can make differential testing inefficient. To address this challenge, we combine the SPARQL syntax and the generated RDF graph to construct syntactically correct and valid SPARQL queries that can return non-empty query results with high possibility. (2) Different RDF stores usually have their own storage and query result formats, which makes the comparison of query results challenging. To address this challenge, we develop a data mapping approach to unify the formats of SPARQL query results in different RDF stores.

To evaluate the effectiveness of RD^2, we apply it on the latest versions of three popular and widely-used RDF stores, i.e., MarkLogic [3], Apache Jena [4], and RDF4j [6]. In total, RD^2 has detected 5 logic bugs in them. We have made RD^2 publicly available at https://github.com/tcse-iscas/RD2.

979-8-3503-2263-7/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-Companion58688.2023.00041

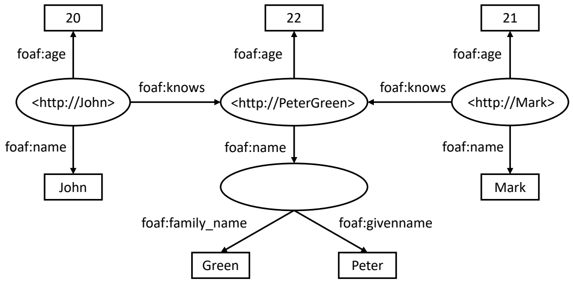

Fig. 2. An RDF graph.

```
1 PREFIX foaf:<http://xmlns.com/foaf/0.1>
2 SELECT ?givenName ?age
3 WHERE {
4         ?person foaf:name ?name .
5         ?person foaf:age ?age .
6         ?name foaf:givenname ?givenName .
7         FILTER (!(?age = 20))
8       }
9 ORDER BY ?givenName
```

Fig. 3. A SPARQL query on the RDF graph in Fig. 2.

II. Preliminaries

RDF triples. An RDF [1] triple can be denoted as $< subject, predicate, object >$. Specifically, $subject$ is an Internationalized Resource Identifier (IRI) or a blank node, $predicate$ is an IRI, and $object$ is an IRI, a literal or a blank node. A literal is a value with a certain data type, e.g., String. A blank node is a node without an IRI.

A set of RDF triples corresponds to an RDF graph. Fig. 2 shows an RDF graph with ten RDF triples. In Fig. 2, an ellipse represents an IRI or a blank node in $subject$ or $object$, a rectangle represents a literal, and a directed edge represents an IRI in $predicate$. For example, we can use `<<http://John>, foaf:age, 20>` to describe a person with name `John` is `20` years old. We can use a blank node to connect the IRI of Peter Green (i.e., `<http://PeterGreen>`) and his given name (i.e., `Peter`) and family name (i.e., `Green`).

SPARQL queries. SPARQL [7] is a standardized query language for RDF stores. Fig. 3 shows a SPARQL query on the RDF graph in Fig. 2, which retrieves the given names and ages of persons whose age is not 20. A SPARQL query generally consists of four components, i.e., a prologue, a SELECT clause, a WHERE clause and an optional solution modifier. Specifically, the prologue declares prefix names, e.g., declaring a prefix name `<http://xmlns.com/foaf/0.1>` as `foaf` (Line 1). The SELECT clause specifies the variables to return (Line 2). The WHERE clause specifies the graph pattern to be queried with triple patterns, FILTER, etc. A triple pattern is similar to an RDF triple except that the $subject$, $predicate$ and $object$ may be a variable. In Fig. 3, the WHERE clause contains three triple patterns and a FILTER expression (Line 3-8). For example, `<?person, foaf:name, ?name>` (Line 4) matches the RDF triples with predicate `foaf:name`, and binds the values of subjects and objects in RDF triples to `?person` and `?name`. The FILTER expression filters persons whose age is not 20. The solution modifier processes the query results by sorting, limiting the number of returned results, etc, e.g., ORDER BY sorts results by `?givenName` (Line 9).

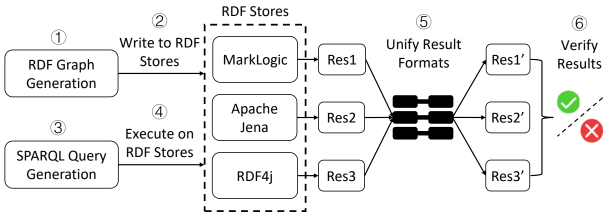

Fig. 4. The overview of RD^2.

III. RD^2

Fig. 4 shows the overview of RD^2. RD^2 first randomly generates an RDF graph and writes it into multiple target RDF stores (Section III-A). Then, RD^2 randomly generates SPARQL queries (Section III-B). Finally, RD^2 runs a SPARQL query on multiple target RDF stores, gets the query result (Res for short in Fig. 4) of each target RDF store, and reveals discrepancies among these query results (Section III-C).

A. RDF Graph Generation

An RDF graph consists of a prefix declaration and a set of RDF triples. The prefix declaration is used to simplify IRIs in each RDF triple, and we generate it by randomly generating a prefix name and its corresponding IRI. Here, the IRI is retrieved from the Python library Faker [21]. An RDF triple consists of a $subject$ (i.e., an IRI or a blank node), a $predicate$ (i.e., an IRI), and an $object$ (i.e., an IRI, a literal, or a blank node). To generate an RDF triple, we first randomly generate a set of IRIs and literals, and then randomly select corresponding data for $subject$, $predicate$, and $object$, respectively. Specifically, we generate IRIs with the Python library Faker [21]. To generate literals, we first randomly choose a data type (e.g., String), and then generate a random value for this data type. After that, we construct an RDF triple by randomly selecting a generated IRI or a blank node for $subject$, randomly selecting an IRI for $predicate$, and randomly selecting an IRI, a literal or a blank node for $object$. Finally, we write the generated RDF graph into target RDF stores.

For example, in Fig. 2, we first generate a prefix name `foaf` for `<http://xmlns.com/foaf/0.1>`. Then, we generate RDF data to construct ten RDF triples. For the first RDF triple, we select an IRI `<http://John>`, an IRI `foaf:age`, and a literal `20` from the previously generated data to construct an RDF triple `<<http://John>, foaf:age, 20>`. For the second RDF triple, we select an IRI `<http://PeterGreen>`, an IRI `foaf:name` and a blank node to construct an RDF triple `<<http://PeterGreen>, foaf:name, [ ]>`.

B. SPARQL Query Generation

To generate syntactically correct and valid SPARQL queries, we build Abstract Syntax Trees (ASTs) to generate SPARQL queries, and choose values for parameters in a SPARQL query from the generated RDF graph to increase the probability of returning non-empty query results.

Generating SPARQL queries. We randomly build a SPARQL query's Abstract Syntax Tree (AST) based on the SPARQL syntax. A SPARQL query consists of a prologue, a SELECT clause, a WHERE clause, and an optional solution modifier. The prologue is used to simplify IRIs in each RDF triple, and we generate it with "PREFIX", a prefix name and its corresponding IRI generated in Section III-A (e.g., Line 1 in Fig. 3). Then we generate the WHERE clause with triple patterns (e.g., Line 4−6 in Fig. 3) and FILTER expressions (e.g., Line 7 in Fig. 3). To generate a triple pattern, we first randomly generate a set of variables, and then randomly select variables to bind to $subject$, $predicate$ or $object$, respectively. We generate a FILTER expression by building an expression tree, which will be explained in the next paragraph. After that, we generate the SELECT clause by randomly selecting several variables (e.g., `?givenName` and `?age` in Line 2 of Fig. 3) existing in the triple patterns of WHERE clause. Finally, we optionally generate a solution modifier with ORDER BY and a variable randomly selected from the triple patterns of WHERE clause (e.g., Line 9 in Fig. 3).

For the FILTER expression in a WHERE clause, we randomly generate it by building an expression tree with a specified maximum depth. The data types of the expression's returning value can be Boolean, String or Numeric. Therefore, we randomly select one of these data types for the root node of the expression tree, and then recursively generate the nodes of the expression tree until the maximum depth is reached. Specially, each generated node must satisfy its required data type. To generate a node, we randomly select an operator or a value that matches its given data type. Leaf nodes can be randomly generated as constants or variables that appear in the query. For the expression $!(?age = 20)$ in Fig. 3, we first randomly choose an operator !, and then randomly create an operator $=$ as its operand. To create the operator $=$, we first select a data type (e.g., Integer), and then generate a variable $?age$ and an Integer value 20 as its operands.

Generating parameter values. Without any guidance, the IRIs and literals in a SPARQL query can be generated randomly. This can cause almost all generated SPARQL queries to return empty results. To tackle this problem, we select IRIs or literals from the generated RDF graphs. Specifically, for the IRI in a triple, we randomly select an IRI from the existing IRIs in the RDF graph. For the constants in a FILTER expression, we randomly select an existing value from the RDF graph, or randomly generate a constant value.

C. Differentially Testing RDF Stores

We adopt differential testing to find discrepancies among multiple target RDF stores. We write the same RDF graph into multiple target RDF stores, and then execute the same SPARQL queries on them. We compare the query results returned by different RDF stores, and find discrepancies among them. The query results can be a list of IRIs, literals or blank node IDs. However, different RDF stores adopt different strategies for generating blank node IDs. Thus, we need to handle this special situation when comparing query results.

Unifying formats of blank node IDs. Since the formats of blank node IDs are different in multiple RDF stores, we need to convert the blank node IDs returned by multiple RDF stores into a unified format. To achieve it, we generate a unique ID *bNodeID* for each blank node when generating RDF graph data. After writing a blank node to an RDF store, we extract its actual ID *actualID* in the RDF store. Then we build a mapping table to record the mapping relationship between *bNodeID* and *actualID* in all target RDF stores. After obtaining the query results, we convert different formats of blank node IDs returned by multiple target RDF stores into IDs in a unified format through this mapping table. For example, the same blank node has different *actualID*s _:bnode7293854958399345579 and $b0$ in MarkLogic and Apache Jena, respectively. We map them into the same $bNodeId$ 1.

IV. Implementation and Usage

We implement RD^2 with around 2,300 lines of Java codes. RD^2 first establishes connections to target RDF stores. For MarkLogic, RD^2 uses MarkLogic APIs to connect the remote MarkLogic Server. For Apache Jena and RDF4j, RD^2 uses Model and Repository APIs to create and visit a local graph database, respectively. Then, RD^2 invokes corresponding APIs provided by each target RDF store to write the generated RDF graph data into target RDF stores, executes the generated SPARQL queries and obtains their returned results.

RD^2 is implemented as a command line tool. After installing target RDF stores, two main steps are needed for running RD^2.

1) **Run RD^2.** We can execute the command "java -jar RD2.jar --`dbname` --`host` --`port` --`username` --`password` --`db-num` --`query-num`" to run RD^2.

    - --`dbname`: The RDF graph database name to test.
    - --`host`: The IP address.
    - --`port`: The port.
    - --`username`: The username to login.
    - --`password`: The password to login.
    - --`db-num`: The number of testing round.
    - --`query-num`: The number of queries in each testing round.

    Note that only MarkLogic needs the first five parameters, since Apache Jena and RDF4j use local graph databases. For example, to test MarkLogic with built-in Apache Jena and RDF4j, we can execute the command "java -jar RD2.jar --`dbname` tmpRDFGraph --`host` 127.0.0.1 --`port` 8000 --`username` root --`password` 123 --`db-num` 10 --`query-num` 100". Then, RD^2 will execute queries on the three RDF stores and report discrepancies among them.

TABLE I
TARGET RDF STORES

RDF Store	DB-Engines Ranking	GitHub Stars	Initial Release
MarkLogic	1	-	2001
Apache Jena	3	890	2000
RDF4j	9	308	2000

2) **Check reports.** A discrepancy report contains the RDF graph, the executed SPARQL queries, and their query results returned by each target RDF store.

V. EVALUATION

A. Methodology

Target RDF stores. We evaluate RD^2 on three widely-used RDF stores, i.e., MarkLogic [3], Apache Jena [4] and RDF4j [6]. Table I shows their DB-Engines ranking of RDF stores [22], GitHub stars and initial releases. We can see that, our experimental subjects are popular and widely-used RDF stores. Among these RDF stores, two of them (i.e., Apache Jena and RDF4j) only support the RDF graph model, while MarkLogic supports multiple models, e.g., RDF and document. We test their latest versions when we started this work, i.e., MarkLogic 10, Apache Jena 3.3.0, and RDF4j 3.6.0.

Testing methodology. We run RD^2 on these three RDF stores for 10 testing rounds. In each testing round, we first randomly generate an RDF graph database with 50 RDF triples, and then randomly generate 1000 SPARQL queries. For each query, we execute it on the three target RDF stores, and verify whether their query results are the same. Any discrepancy can be reported by RD^2. For the reported discrepancies, we manually simplify and analyze them to figure out which RDF store performs unexpectedly and whether it can be identified as a potential bug. After filtering out duplicated bugs, we submit a bug report for each bug to the corresponding community on GitHub or Stack Overflow.

B. Detection Results

RD^2 takes about 94.5 seconds to run each testing round. In each testing round, 30% of SPARQL queries return non-empty results on average. For 10 testing rounds, RD^2 reports 445 discrepancies in total. After filtering out duplicated discrepancies, we finally find 6 discrepancies among the three RDF stores. After analyzing these discrepancies, we identify 5 logic bugs from 5 discrepancies, and the remaining one discrepancy is caused by different SPARQL versions that these RDF stores support. We have submitted these bugs to corresponding developers. At the time of writing this paper, 2 bugs have been confirmed, and the remaining 3 bugs have been considered as intended behaviors due to different implementations of RDF stores. The root causes of these bugs mainly relate to automatic type conversion, numeric representation, and handling the same values with different data types.

MarkLogic. We find four logic bugs in MarkLogic. In the first bug, MarkLogic cannot convert a value with Integer type to Double type automatically, which leads to decimal overflow in FILTER expressions. In the second bug, MarkLogic omits equal values with different data types. In the third bug, MarkLogic treats the same value with different data types as equivalent, which does not follow the RDF specification. In the fourth bug, MarkLogic returns literals with different numeric representative format from the other two RDF stores. For example, MarkLogic returns 1589301716 while Apache Jena and RDF4j return $1.589301716E9$ for the same literal. MarkLogic developers explain that the scientific notation appears to be lost for Double in JSON.

RDF4j. We find one logic bug in RDF4j. RDF4j cannot automatically perform type conversion between Integer and its derived types, e.g., Int and Long. This prevents users from retrieving data with derived types of Integer using the equal values with Integer type. RDF4j developers have prepared a feature proposal to fix this bug.

VI. RELATED WORK

Testing of graph database systems (GDBs). Some approaches [11]–[13], [23] are proposed to test GDBs. Grand [11] and GDsmith [12] apply differential testing [24] to detect logic bugs in Gremlin-based and Cypher-based GDBs, respectively. GDBMeter [13] adopts metamorphic testing by partitioning a given query into three derived sub-queries to test GDBs. However, all of these works cannot detect bugs in RDF stores due to different graph models and query syntaxes.

Testing of relational database systems (RDBMSs). Many approaches [14]–[20], [25]–[33] are proposed to find bugs in RDBMSs that use SQL as a standardized query language. RAGS [14], APOLLO [31] and TAQO [32] utilize differential testing for detecting bugs in RDBMSs. SQLsmith [33] is used to detect bugs causing exceptions or crashes in RDBMSs. TLP [15], NoREC [16], PQS [17] and DQE [18] develop various test oracles to detect logic bugs and optimization bugs, and have found many bugs in popular RDBMSs. DT2 [19] and Troc [20] detect transaction bugs in RDBMSs. However, these tools cannot be directly applied to test RDF stores because SPARQL has different data models and query syntaxes from SQL.

VII. CONCLUSION

We present RD^2, an automated testing technique for revealing discrepancies among RDF stores and detecting logic bugs in RDF stores. RD^2 randomly generates an equivalent RDF graph for multiple RDF stores, and checks whether these RDF stores can return the same query result for a given SPARQL query. Our experiment shows that RD^2 has detected 5 logic bugs in three widely-used RDF stores, and 2 bugs have been confirmed by developers.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (62072444), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences.

References

[1] I. Abdelaziz, E. Mansour, M. Ouzzani, A. Aboulnaga, and P. Kalnis, “Query optimizations over decentralized RDF graphs,” in *Proceedings of International Conference on Data Engineering (ICDE)*, 2017, pp. 139–142.

[2] W. Ali, M. Saleem, B. Yao, A. Hogan, and A. N. Ngomo, “A survey of RDF stores & SPARQL engines for querying knowledge graphs,” *The VLDB Journal*, vol. 31, no. 3, pp. 1–26, 2022.

[3] “MarkLogic,” https://www.marklogic.com/, 2022.

[4] “Apache Jena,” https://jena.apache.org/, 2022.

[5] “GraphDB,” https://www.ontotext.com/, 2022.

[6] “RDF4j,” https://rdf4j.org/, 2022.

[7] J. Pérez, M. Arenas, and C. Gutierrez, “Semantics and complexity of SPARQL,” *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 3, pp. 1–45, 2009.

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

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

[10] T. Berners-Lee, J. Hendler, and O. Lassila, “The semantic web,” *Scientific american*, vol. 284, no. 5, pp. 34–43, 2001.

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

[12] Z. Hua, W. Lin, L. Ren, Z. Li, L. Zhang, W. Jiao, and T. Xie, “GDsmith: Detecting bugs in Cypher graph database engines,” 2023.

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

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

[15] M. Rigger and Z. Su, “Finding bugs in database systems via query partitioning,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020.

[16] ——, “Detecting optimization bugs in database engines via non-optimizing reference engine construction,” in *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 1140–1152.

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

[18] J. Song, W. Dou, Z. Cui, Q. Dai, W. Wang, J. Wei, H. Zhong, and T. Huang, “Testing database systems via differential query execution,” in *Proceedings of International Conference on Software Engineering (ICSE)*, 2023.

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

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

[21] “Python faker,” https://github.com/joke2k/faker, 2022.

[22] “DB-Engines,” https://db-engines.com/en/ranking/rdf+store, 2022.

[23] W. Lin, Z. Hua, L. Zhang, and T. Xie, “GDiff: Automated differential performance testing for graph database systems,” in *Proceedings of International Conference on Software Engineering (ICSE)*, 2023.

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

[25] J. Ba and M. Rigger, “Testing database engines via query plan guidance,” in *Proceedings of International Conference on Software Engineering (ICSE)*, 2023.

[26] J. Liang, Y. Chen, Z. Wu, J. Fu, M. Wang, Y. Jiang, X. Huang, T. Chen, J. Wang, and J. Li, “Sequence-oriented DBMS fuzzing,” in *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2023.

[27] Z. Jiang, J. Bai, and Z. Su, “DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation,” in *Proceedings of USENIX Security Symposium (USENIX Security)*, 2023.

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

[29] Y. Liang, S. Liu, and H. Hu, “Detecting logical bugs of DBMS with coverage-based guidance,” in *Proceedings of USENIX Security Symposium (USENIX Security)*, 2022, pp. 4309–4326.

[30] M. Wang, Z. Wu, X. Xu, J. Liang, C. Zhou, H. Zhang, and Y. Jiang, “Industry practice of coverage-guided enterprise-level DBMS fuzzing,” in *Proceedings of IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP)*, 2021, pp. 328–337.

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

[32] Z. Gu, M. A. Soliman, and F. M. Waas, “Testing the accuracy of query optimizers,” in *Proceedings of International Workshop on Testing Database Systems*, 2012, pp. 1–6.

[33] “SQLsmith,” https://github.com/anse1/sqlsmith, 2022.