Finding Logic Bugs in Graph Stores via Label Partitioning

Matteo Kamm
ETH Zurich
Switzerland
matkamm@student.ethz.ch

Mike Marti
ETH Zurich
Switzerland
mikmarti@student.ethz.ch

ABSTRACT

Graph databases store data as properties of graph structures and allow nodes and their relationships to be queried efficiently. With the rise of social networks and big data, graph databases have gained popularity because they allow the storage and querying of highly-connected data. Like with traditional database software, such as Relational Database Management Systems, graph databases can be affected by logic bugs, causing them to compute an incorrect result for a given query. Detecting faults and ensuring the correctness of such Graph Database Management Systems is of utmost importance. In this report we present a new metamorphic testing approach called Label Partitioning which we implemented for the Graph Database Management System Neo4J. The idea of Label Partitioning is to select nodes and their neighborhood based on an identifier in an initial query. Individual queries are then performed to select proper disjoint subsets of the initial query result to see whether the partitioning invariant holds for the Database Management System. So far our prototype has not found any bugs. We believe that by adjusting the parameters or by refining the current oracle, e.g. by adding `WHERE` clauses to increase the query complexity, this approach has the potential to find a multitude of bugs in Graph Database Management Systems.

1 INTRODUCTION

Graph Database Management Systems (GDBMS) introduce a novel way of storing data as part of graph structures. In recent years, the popularity of such systems has increased drastically due to their applicability in social networks and big data. One such well-established Java based implementation is Neo4J. Neo4J is based on the so called Property Graph Model which organizes data as part of nodes, relationships and properties. Nodes hold data in the form of key-value pairs (called properties). Furthermore, nodes can be assigned multiple labels that are used to classify them. A core concept of graph databases are the directed edges between nodes (also called relationships). In most applications those relationships are the main entity that one wants to express using a graph database. Similarly to nodes, relationships can also have properties and labels. In Figure 2a a simple graph modelling the entities person, technology and company as well as their relationships is depicted. For instance, the person with the name property Jennifer likes the technology of type graphs and she works for the company named Neo4J. Michael works for the same company as Jennifer until the year 2022.

Previous work has proposed oracles specific to RDBMSs and found a substantial amount of bugs. However, those do not utilize invariants present only in graph datases for testing. This lead us to believe that, by using graph database invariants, we would be able to find similar faults. As part of our work, we introduce the software prototype GDBLancer that is able to test the Neo4J graph database via the Label Partitioning test oracle. The purpose of this prototype is to find logic bugs in the query processor that leads to incorrect result sets. We consider a result set to be incorrect, if a record is not fetched even though it should be returned by the query or if a record is incorrectly included.

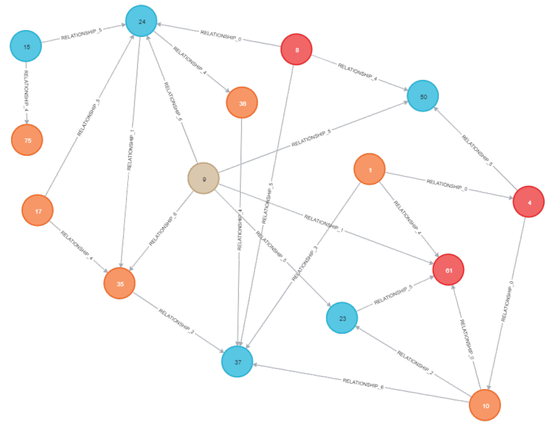
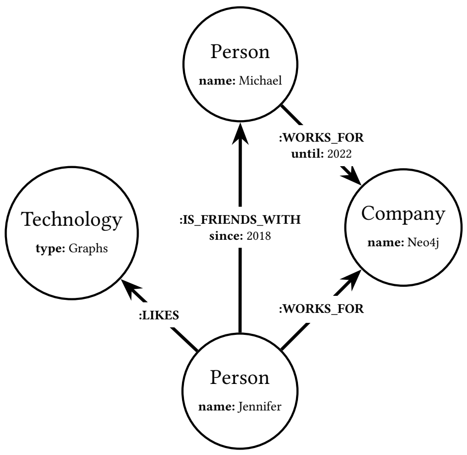


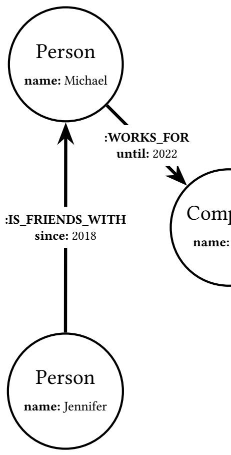
Figure 1: Example of a graph generated by GDBLancer

The core idea of the Label Partitioning oracle is to have a method that focuses on testing the correct selection of connected nodes in the graph stored by the database. The reason for our focus on this specific part of graph databases, is because it is one of the main features and advantages of GDBMSs compared to traditional RDBMSs. Therefore, it is of high importance that this frequently used part of the Database Management System is free of errors.

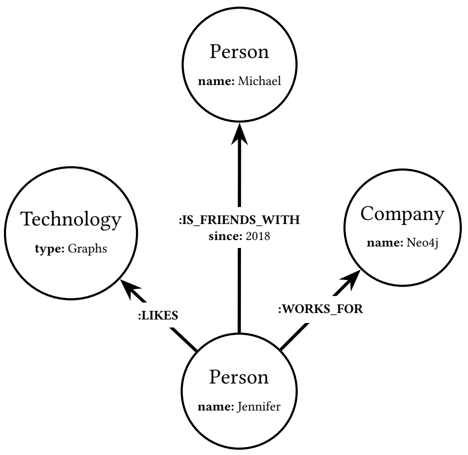
To get a better understanding of how the oracle works, we will briefly explain it using an example. Consider a randomly chosen label $l \in L$ where L is the set of all available labels of the graph. The oracle executes a query Q, which computes a set of all nodes V_l with label l as well as their incident relationships and neighboring nodes. We call this result set $RS(Q)$. In our example $l = Person$ and Figure 2a depicts the result set $RS(Q)$. In that case V_l is made up of the nodes representing Michael v_M and Jennifer v_J, i.e. $V_l = \{v_M, v_J\}$. The oracle then computes the neighbors and incident relationships for each node $v \in V_l$ individually, which gives us the result sets $RS(Q_{v_M})$ and $RS(Q_{v_J})$ for both v_M and v_J respectively. A depiction of their result set can be seen in the Figures 2b and 2c. It then uses the union operator $\cup$ to determine whether or not $RS(Q_{v_M}) \cup RS(Q_{v_J}) = RS(Q)$. If this equivalence does not hold, there is at least one query Q' that collected either too many or not enough neighboring nodes / incident relationships, which means that the query processor of the GDBMS that is being tested has a bug when executing query Q'.



(a) Result of query Q.



(b) Result of query Q_{v_M}.



(c) Results of query Q_{v_J}.

Figure 2: Results of the example queries.

To have an executable implementation of the Label Partitioning test oracle, we developed the GDBMS testing infrastructure GDBLancer. Its purpose is to generate a random graph, store this graph in a graph database and then execute the user specified test oracle. Generating random graphs can be achieved as follows: First generate a set of labels, relationship types and a unique set of available properties for each label and relationship type respectively. These sets are then used as input to generate a random in-memory graph. At this point the graph is still independent of the GDBMS and therefore it would be possible to store the graph on different graph database implementations. An example of such a random graph saved on a Neo4J database can be seen in Figure 1.

We have executed the Label Partitioning oracle for several hours on different graph sizes using the GDBLancer testing infrastructure. So far, no bugs have been found. We are, however, optimistic that by running the oracle for an extensive period of time, by applying small tweaks to the oracle itself and by adding support for other Graph Database Management Systems, the Label Partitioning oracle has the potential to find a multitude of bugs.

Metamorphic test oracles [2] that are based on the concept of Query Partitioning [8] have already been used to successfully find logic bugs in RDBMSs. However, the detection of logic bugs in GDBMSs has not yet been researched in great detail and existing papers do not use a Query Partitioning approach for their test oracles. In summary, this paper establishes the following new concepts:

- GDBLancer, a lightweight and extensible GDBMS testing infrastructure.
- Label Partitioning, a GDBMS specific testing oracle that is based on the concept of Query Partitioning.
- An evaluation of how Label Partitioning can be expanded to find logic bugs in GDBMSs.

2 BACKGROUND

Graph Database Management System. Graph Database Management Systems [3, 7, 9] have become more popular with the rise of social networks and big data in recent years. Storing data as part of graphs can be a more optimal way to model data storage for certain applications. Another key advantage over traditional data storage methods such as relational tables, is that related data might be processed with higher performance. GDBMSs are often schema-less, meaning that the data does not have to adhere to a fixed strucutre. This allows software systems to evolve over time without having to think about the schema changes and data migrations. In the model of graph databases, the data is stored as part of the nodes and edges that describe relationships between nodes. Contrary to Relational Database Management Systems, where data related to the connection of two entities has to be modeled as an intermediate table, Graph Database Management Systems treat edges as first-class citizens, meaning that data can be directly stored as part of an edge itself. Instead of SQL, GDBMSs use domain specific query languages with a syntax that allows for easy selection of nodes and their corresponding relationships.

Neo4J. Neo4J [9] is a widely used Graph Database Management System implemented in Java and is based on the Property Graph Model. In this model the data is organized as nodes, relationships and properties. Nodes are tagged with labels that can be used to describe their roles in the problem domain. Attached to nodes are properties in the form of key-value pairs. Relationships connect two nodes and have a direction as well as a type. Just like nodes, relationships can have properties in the form of key-value pairs. The standardized query language used by Neo4J is called Cypher [4]. Cypher is based on ASCII art which makes the syntax easy to read and understand. Furthermore, Cypher is a so called declarative query language which means that a user simply describes what he wants a query to return / update and not how the data should be retrieved / updated. So far GDBLancer only supports Neo4J but it was written in a generic way and it is possible to support other GDBMSs.

Metamorphic testing. Metamorphic testing [2] is a testing technique that can be used to generate new testcases based on a metamorhpic relation and previously known inputs and outputs of a system. An example for such a relation for a program that calculates the sine value is $\sin(x) = \sin(x + 2\pi)$. One can generate a follow-up test case to establish that the relation holds based on the result of a previous test case. The Label Partitioning oracle proposed in our work uses the metamorphic testing approach by first generating some query and computing its result set. It then splits the query into multiple smaller ones and expects the union of their disjoint result sets to be the same as the initial result set.

Query Partitioning. Query Partitioning [8] is a technique to find logic bugs in Database Management Systems. The main idea is to derive n individual queries $Q_1, \ldots, Q_n$ from a given query Q and a corresponding result set $RS(Q)$, each of which computes partial result $RS(Q_i)$. Using a predefined composition operator $\bullet$, it is then possible to check the equivalence of the composed partial results and $RS(Q)$, i.e. $RS(Q_1) \bullet \ldots \bullet RS(Q_n) = RS(Q)$. Whether or not this is the case determines the correctness of the Database Management System. The goal is to find queries that trigger different execution strategies to find inconsistencies in the query processor of the Database Management System.

3 APPROACH

This section contains information on how GDBLancer uses the Label Partitioning test oracle to find bugs in GDBMSs.

3.1 Overview

Figure 3 illustrates the steps of running the Label Partitioning oracle. First, it creates a random in-memory graph with random nodes, relationships, labels and relationship types (see step ①). Based on the label of a node, we then assign random properties (key-value pairs) to nodes. This is to ensure that a certain set of properties is available on nodes with a fixed label. In step ② the in-memory graph data structure is saved as a graph into a database. Only now does the software become Graph Database Management System specific. The following steps ③ to ⑤ execute the Label Partitioning on the Neo4J database. A label l is chosen at random and all nodes V_l with label l are selected. Then all relationships R that are incident to any vertex of V_l and all neighboring nodes N of V_l are queried. So far the oracle performed the initial queries that are needed to establish the expected behaviour. Step ⑤ can be seen as the partitioning step where for each pair of $N \times R$ we select their relationship and remove the found nodes from N and V_l with respect to multiplicity. Finally, the sets N and R should be empty and there should not have been an attempt to remove any node or relationship too often.

3.2 Random Graph Generation

In step ① a random in-memory graph is generated. This generation happens in two steps: First a graph structure exhibiting several predefined properties is created, then a random in-memory graph is generated based on that structure. Such a structure contains the following important components:

- A set of random labels L.
- A mapping λ between labels and sets of random unique property types. This ensures that nodes with a label can contain only a finite set of unique properties.
- A set of relationship types R_T.
- A mapping ρ between relationship types and sets of random unique property types. This again ensures that relationships of a type are only assigned unique properties of a finite set.

In a next substep the random graph is generated in-memory. In order to do so we first determine the amount of nodes n as a random integer in a configurable interval $[n_l, n_u]$. Then for each of the n nodes exactly one label out of L is selected at random. Based on the selected label l the set of assignable properties $\lambda(l)$ is determined and a subset of those is picked and instantiated with random values. For each pair of nodes of $V \times V$ an edge gets created with a configurable probability. Each edge e of the in-memory graph is assigned a random relationship type $r \in R_T$. Based on r a subset of $\rho(r)$ is selected and instantiated to determine the properties of the edge e.

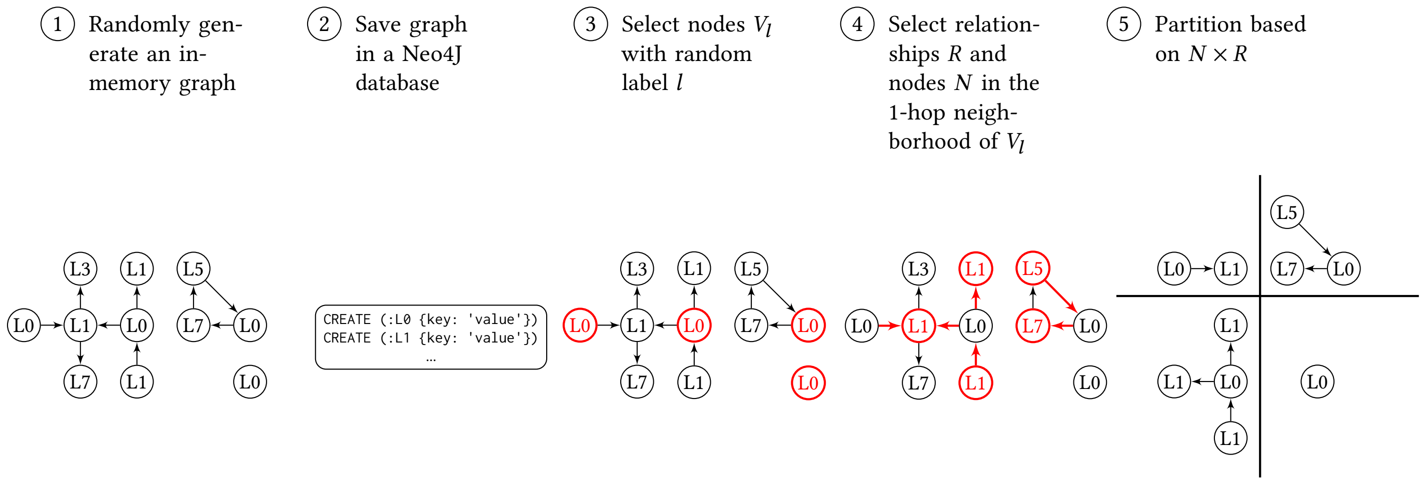
3.3 Neo4J Generation

Up until now the implementation is GDBMS independent meaning that this code can be reused for any graph database. This would allow GDBLancer to be extended to support other GDBMSs in the future. See section 4 for more details. In step ② we generate Neo4J specific queries to insert the in-memory graph into an actual graph database. To do so, the nodes are first serialized as property strings and inserted one by one into the database. Neo4J assigns a unique id to each node which will be used to then generate the edges on the database. Relationships are serialized in a similar fashion and the endpoints of an edge are identified based on their Neo4J id.

3.4 Label Partitioning

Steps ③ to ⑤ perform the Label Partitioning. Note that the steps ① and ② were independent of the testing method and are executed even if a different kind of oracle is chosen. It would even be possible to change the testing oracle at runtime to trigger possible bugs on the System Under Test. Label Partitioning proceeds in the following important steps:

(1) First a random label $l \in L$ is chosen and the nodes with label l are selected. We call the set of all these nodes V_l.
(2) Then the 1-hop neighborhood of the nodes with label l is selected. This gives us a list of ids of nodes and relationships that are contained within this neighborhood. This list of ids respects multiplicity meaning that nodes (relationships) that adjacent (incident) to two or more nodes with label l are counted multiple times. An example of this can be seen in Figure 3, where the node in the middle with label L1 is the neighbor of two nodes with the selected label L0. As a consequence, nodes with label l can also be part of this 1-hop neighborhood and therefore it is also possible to have ids of such nodes in the list.
(3) For each node in V_l we iterate over all relationships types individually and issue a query, selecting the ids of all incident edges of the currently processed relationship type, as well as the ids of the adjacent nodes connected by these edges. The selected ids are then removed from the list of ids computed in the second step.


1 Randomly generate an in-memory graph
2 Save graph in a Neo4J database
3 Select nodes V_l with random label l
4 Select relationships R and nodes N in the 1-hop neighborhood of V_l
5 Partition based on $N \times R$
L0
L1
L3
L5
L7

```
CREATE (:L0 {key: 'value'})
CREATE (:L1 {key: 'value'})
…
```

Figure 3: Overview of the approach implemented in GDBLancer.

(4) After performing the previous step on all nodes with label l, all neighbors and relationships of the 1-hop neighborhood should have appeared exactly as often as they were selected in the second step. The list of ids is expected to be empty and there should have never been an attempt to remove an id that was not present in the list.

3.5 Technical difficulties

The main technical difficulty occured during the early stage of the development phase. Neo4J can be used in two different ways: As an external server which is accessed over a binary protocol supported by a Java driver or as an embedded server which runs in the same JVM as the application accessing it. As the name suggests, the embedded version is useful for systems where the database and application run on the same device and no other application needs to access the graph database. The problem is that these two versions have entirely different APIs and cannot be exchanged freely. This meant that we had to decide on the version we want to use early on without knowing about the possible consequences. In the end, we decided to use the embedded version as it does not require the user to setup an external database which is convenient. One downside of this decision was that the API description was lacking a lot of cruicial information and it was challenging to access the embedded database itself to investigate whether our application worked as intended. To do so we had to copy the generated database files into a directory and import the database into an existing Neo4J server.

4 IMPLEMENTATION

We implemented GDBLancer in Java and 980 lines of code. In this section we explain the most significant implementation decisions taken when developing the GDBLancer application.

Extensibility. One of the main non-functional requirements that we tried to satisfy during the development of GDBLancer was the extensibility with regards to adding support for additional GDBMSs and new test oracles. To achieve this, instead of directly generating the database graph, we implemented an in memory graph datastructure that supports the most important features found in property graphs. Due to this design decision, it was possible to completely decouple the random generation of the graph and all database specific parts of the application. The oracle implementations are decoupled from the driving code that calls them via interfaces.

Logging. To be able to recreate bugs found by the oracle, the GDBLancer application uses a logging library to save all executed queries. The oracles perform a lot of queries during their execution, which reduces the performance of the application and increases the size of the logfile significantly. To mitigate this performance issue, we allow the user to specify whether the system should log all messages or only the important queries i.e. the graph creation queries and those that trigger a bug. In order to do this we buffer logged messages in an intermediate data structure. Calling the logging framework only happens when an exception is thrown and the appropriate reporting flag is set. This gives us fine-grained control over which queries are written to logfiles and which can be safely ignored.

Probabilities and bounds. The graph generation process uses constant probabilities, as well as lower and upper bounds. These values are used to, for example, give a bound on how many nodes the graph should have or specify how prevalent edges should be. To be able to easily manipulate them during the evaluation phase, we decided to declare them as constants inside a dedicated class.

Supported property types. Neo4J supports different data types for the values of properties. GDBLancer supports `long`, `String` and `double` properties. The reason why we support these three types in particular is because they are easy to generate random values for and because they are among the most basic types, which makes them likely to be available in all GDBMSs.

5 RESULTS

As part of this project, we implemented the prototype GDBMS testing infrastructure GDBLancer. This infrastructure includes the Label Partitioning test oracle. During the evaluation process we executed the Label Partitioning oracle on a Neo4J database using three different random graphs, each of which was generated using different probabilities and bounds. All of these executions ran for about 3 hours each.

Infrastructure. The experiments were conducted using a computer with a 4-core Intel i7-4790K CPU at 4.00 GHz and 16 GB of DDR3 memory at 1600 MHz running on an Archlinux 5.12.6 distribution. We used the embedded Neo4J database with version number 4.2.5 for all experiments.

Bugs found. So far, no bugs have been found. We assume that this is due to one or a combination of the following reasons:

- Neo4J is one of the most popular Graph Database Management Systems. As such, a lot of bugs are already found by their enormous user base. As a result, the query processor of Neo4J is probably one of the most robust when it comes to GDBMSs, which means that finding new bugs is challenging and improbable.
- The Label Partitioning test oracle in its implemented form in the GDBLancer prototype is unlikely to trigger different execution strategies during the evaluation of its queries. This is because we use the default generated identifier in the Label Partitioning oracle, which by default is not indexed in Neo4J. If this execution strategy is faulty, it will perform the error in both the original query, as well as the partial queries. Even though the result sets are wrong, they will be equivalent and the the oracle therefore does not detect this bug.
- Due to the limited time frame of this project, we were not able to run the oracle as much and as long as we would have wanted to. It is possible that the current implementation of the Label Partitioning test oracle is able to find bugs when it is executed for a longer duration of time.
- GDBLancer only generates one graph database per run and this might limit the search space drastically. Perhaps, it would therefore be beneficial to generate multiple databases per execution although this would decrease the performance of the application.

6 RELATED WORK

This section contains information about work that is related to the approach presented in this paper.

Metamorphic testing of DBMSs. There has already been research conducted in the field of automatic metamorphic testing of Relational Database Management Systems. One such example is SQLancer [8]. SQLancer uses a metamorphic testing approach to detect logic bugs in the query processor of various RDBMSs. It is also the main inspiration for our work. The concept of Query Partitioning, which the Label Partitioning oracle is based upon, was first proposed in their paper. They used the concept to develop the Ternary Logic Partitioning (TLP) oracle, a basic version of which we also implemented in a first iteration of GDBLancer. They were able to detect over 100 bugs using this approach in multiple different RDBMSs. This is also the reason why we believe that their technique can be effective when applied to GDBMSs.

Model-based testing of GDBMSs. There has already been research conducted in the field of model-based testing of graph databases [1, 5]. These systems are not only able to find functional faults in the queries they execute, but they can also detect errors in the evaluation mechanism of the graph database itself. The downside to this approach is that model-based testing requires a system and a model describing that system. It is thus more tailored towards testing the database logic of business applications rather than testing the graph database system itself.

Random generation of graphs. Some approaches have been proposed to speed up the process of generating graph structures [6]. These algorithms highly benefit from multicore parallelism and GPU optimized code. The current graph generator used in GDBLancer is rather simplistic and was not optimized in any way. Using a more efficient algorithm would allow us to generate larger graphs and certain bugs might show themselves only in conjunction with very large data sets.

7 CONCLUSION

In this paper we have presented a novel testing method called Label Partitioning to find logic bugs in the query processor of Graph Database Management Systems. This testing oracle builds on the concept of Query Partitioning, where an existing query is split into multiple individual ones, each of which computes a disjoint subset of the original result set. The disjoint union of the result sets of these queries is then expected to be the same as the result set of the initial query. Label Partitioning heavily relies on the fact that connected data can easily be queried in graph databases. In addition, we present a testing infrastructure called GDBLancer, which consists of a random graph generator and an implementation of the Label Partitioning oracle. The application is implemented in a way that allows for high extensibility with regards to new GDBMSs and test oracles. For instance, one could save the in-memory graph representation to a different GDBMS with minimal effort, since the core of the application is written in a generic fashion.

The implemented infrastructure in its current state is a proof of concept and only tests Neo4J. Adding support for additional GDBMSs, introducing new test oracles and improving the performance through parallelism would be prime examples of how the current system can be improved upon. Similarly, the current implementation of the Label Partitioning test oracle can be expanded by adding constraints on the selected nodes and edges to make the queries more complex and thus cover a wider range of possible query executions.

ACKNOWLEDGMENTS

We thank Manuel Rigger for his support and helpful feedback throughout the entirety of the project.

REFERENCES

[1] Raquel Blanco and Javier Tuya. 2015. A test model for graph database applications: an MDA-based approach. In *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*. 8–15.

[2] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

[3] Facebook, Inc. 2021. GraphQL. Working Draft, May. 2021. Online at https://spec.graphql.org/.

[4] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*. 1433–1445.

[5] Leen Lambers, Sven Schneider, and Marcel Weisgut. 2020. Model-Based Testing of Read Only Graph Queries. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 24–34.

[6] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. 2011. Fast random graph generation. In *Proceedings of the 14th international conference on extending database technology*. 331–342.

[7] Rob Reagan. 2018. Cosmos DB. In *Web Applications on Azure*. Springer, 187–255.

[8] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[9] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.".