Implementing UNIQUE Integrity Constraint in Graph Databases

Kornelije Rabuzin, Martina Šestak, Mladen Konecki
Faculty of Organization and Informatics, Varaždin
University of Zagreb
Zagreb, Croatia
e-mail: kornelije.rabuzin@foi.hr, msestak2@foi.hr, Mladen.Konecki@foi.hr

Abstract—**Databases enable users to store and retrieve data. However, uncontrolled data entry often results in having duplicate or incorrect data stored in the database, which makes it inconsistent. To prevent this, it is recommended to specify integrity constraints in the database. In this paper, the current possibilities in the field of integrity constraints, with special emphasis on graph databases as a relatively new category of NoSQL databases, are discussed. After giving an overview of the current situation regarding integrity constraints in mostly used graph database query languages, a successful implementation of UNIQUE integrity constraint in a Neo4j graph database is shown.**

Keywords-integrity constraints; graph databases; Gremlin; UNIQUE

I. INTRODUCTION

Inserting data into a database is an important process that must be properly managed and controlled in order to ensure the validity of the inserted data (no Garbage-In-Garbage-Out situations) and to enforce database integrity, i.e., to maintain the database in a consistent state. The database is considered to be consistent if it satisfies the set of specified integrity constraints [1]. Formally, integrity constraints can be defined as formal representations of invariant conditions for the semantic correctness of database records [2] and as general statements and rules that define the set of consistent database states or changes of states, or both [3].

Thus, integrity constraints contain the semantic information about data stored in the database, i.e., they are properties that must be satisfied for the database and data we want to insert into the database. If these properties are satisfied, then the data is considered to be semantically correct and the operation or transaction is executed successfully. Otherwise, if the integrity constraint is violated, then the transaction is rejected or a specific compensation action is activated, which will reduce the impact of that transaction and repair the current database state.

Once constraints are specified, the database management system (DBMS) has to ensure that all constraints are satisfied and none are broken. Eventually, it is possible that some constraints will be broken during a transaction, but when the transaction ends, all constraints have to be satisfied.

Nowadays, most relational DBMSs provide some kind of support for declarative integrity constraints, which can be grouped into three categories:

- Column constraints, which are specified on table columns (e.g., NOT NULL, UNIQUE, CHECK, PRIMARY KEY, REFERENCES);
- Table constraints, which are used when some constraints cannot be specified as column constraints (e.g., when tables have compound primary keys consisting of multiple columns, then one cannot specify a PRIMARY KEY column constraint on these columns, since the PRIMARY KEY clause can appear only once within the table definition); and
- Database constraints, which are defined for multiple tables or the entire database through assertions, which belong to the database schema and can be created using the CREATE ASSERTION clause.

Triggers represent an interesting alternative for specifying more complex constraints involving several tables, i.e., database constraints. Basically, when an event like INSERT or UPDATE occurs in the database, a function (procedure) is activated and several different statements can be executed as a reaction to the event.

Unfortunately, most DBMSs are quite limited when it comes to expressing more complex conditions and rules to be satisfied, but also the compensating actions responsible for repairing the database state. This disadvantage can be replaced by expressing integrity constraints as triggers and stored procedures. However, note that they are more challenging to manage as data and constraints evolve.

Lately, maintaining database integrity has become very costly and time consuming due to the increasing amount of data stored in databases and the large number of specified integrity constraints, where each requires some time to be validated.

In the last few years, new database solutions have appeared on the database market as an alternative to traditional relational databases. These solutions avoid using the Structured Query Language (SQL) as the only query language for interacting with databases, so they are known under the term Not only SQL (NoSQL) databases. NoSQL databases can be classified in four solution groups: key-value databases, document databases, column-oriented databases, and graph databases.

Unlike relational databases, NoSQL databases are usually schema-less, thus not placing much attention and importance on strictly maintaining database consistency.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-513-5

As already mentioned, graph databases represent a special category of NoSQL databases. Even though they are a relatively new alternative to relational databases, much effort has been made in their development (both in graph DBMS products implementation and the literature). According to [4], Neo4j, the most widely used graph DBMS, is the 21st most popular DBMS on the database market (including relational and other NoSQL DBMSs) and has constant growth in popularity.

Like every database, graph databases are based on the graph data model, which consists of nodes connected by relationships. Each node and relationship contains (not necessarily) properties, and is given a label. Hence, data is stored as property values of nodes and relationships. In the graph data model, nodes are physically connected to each other via pointers (this property is called index-free adjacency [5], and the graph databases that implement index-free adjacency are said to be using native graph processing), thus enabling complex queries to be executed faster and more effectively than in a relational data model.

The main advantage of graph databases is their ability to model and store complex relationships between real-world entities. Nowadays, there are some situations where it is easier to model a real-world domain as a graph, rather than as a set of tables connected via foreign keys in the relational data model. Querying a graph database is much faster (especially as the database size grows) when nodes are connected physically as compared to relational databases, where many performance-intensive table join operations must be made. Except for the performance improvements, graph databases offer much bigger flexibility in data modelling, since no fixed database schema must be defined when creating a graph database. The lack of a fixed schema makes later changes to the database structure much simpler, since graphs can be easily updated by adding new subgraphs, nodes, relationships, properties or labels.

In this paper, we discuss integrity constraints in graph databases. Section 2 contains an overview of graph databases, related researches on the topic of integrity constraints in graph databases and the current level of support for integrity constraints provided by most commonly used graph DBMSs. In Section 3, the concrete implementation of the UNIQUE integrity constraint in a Neo4j graph database is shown and explained. Finally, in Section 4, we give a short conclusion about the topic of this paper and provide some brief information about our future work.

II. INTEGRITY CONSTRAINTS IN GRAPH DATABASES

When it comes to data consistency and integrity constraints in graph databases, one can notice that this area is still not developed in detail and provides space for further improvements and research. Some people even say that the reason for this is the flexible and evolving schema supported by graph databases, which makes integrity constraint implementation more difficult.

As discussed in [6], Angles and Gutierrez wrote a research paper in which they identified several examples of important integrity constraints in graph database models, such as schema-instance consistency (the instance should contain only the entities and relations previously defined in the schema), data redundancy (decreases the amount of redundant information stored in the database), identity integrity (each node in the database is a unique real-world entity and can be identified by either a single value or the values of its attributes), referential integrity (requires that only existing entities in the database can be referenced), and functional dependencies (test if one entity determines the value of another database entity).

In [7], Angles also considered some additional integrity constraints such as types checking, verifying uniqueness of properties or relations and graph pattern constraints.

Apart from the Neo4j graph DBMS, which will be used for UNIQUE integrity constraint implementation, there are other graph DBMSs available on the database market. In this paper, an overview of the support level for integrity constraints will be given for the five most popular graph DBMSs. According to [8], when it comes to graph DBMS popularity ranking, Neo4j DBMS is followed by Titan, OrientDB, AllegroGraph and InfiniteGraph. The level of support in Neo4j DBMS will be explained in the following subsections.

Titan is a graph DBMS developed by Aurelius, and its underlying graph data model consists of edge labels, property keys, and vertex labels used inside an implicitly or explicitly defined schema [9]. After giving an overview of its characteristics and features, one can say that the level of support for integrity constraints is pretty mature and developed. Titan offers the possibility of defining unique edge and vertex label names, edge label multiplicity (maximum number of edges that connect two vertices) and even specifying allowed data types and cardinality of property values (one or more values per element for a given property key allowed). OrientDB is a document-graph DBMS, which can be used in schema-full (all database fields are mandatory and must be created), schema-hybrid (some fields are optional and the user can create his own custom fields) and schema-less (all fields are optional to create) modes [10]. OrientDB provides support for defining and specifying even more integrity constraints, such as:

- Defining minimum and maximum property value;
- Defining a property as mandatory (a value for that property must be entered) and readonly (the property value cannot be updated after the record is created in the database);
- Defining that a property value must be unique or cannot be NULL;
- Specifying a regular expression, which the property value must satisfy; and
- Specifying if the list of edges must be ordered.

Unlike Titan and OrientDB, AllegroGraph does not provide support for any kind of user-defined integrity constraints, which means that there are no database control mechanisms to verify the validity of the inserted data. AllegroGraph databases only ensure that each successfully executed database transaction will change the database's consistent internal state [11]. InfiniteGraph is a distributed

graph database solution offering strong or eventual database consistency, which only supports property value type checking and referential integrity constraints [12].

As already mentioned, Neo4j is the most commonly used graph DBMS, so its support for integrity constraints will be discussed by giving a practical overview of features provided by query languages used in a Neo4j database: Cypher and Gremlin. In the following subsections, their characteristics and the level of support for integrity constraints will be reviewed.

A. Cypher

Cypher is a declarative, SQL-like query language for describing patterns in graphs using ascii-art symbols. It consists of clauses, keywords and expressions (predicates and functions), some of which have the same name as in SQL. The main goal and purpose of using Cypher is to be able to find a specific graph pattern in a simple and effective way. Writing Cypher queries is easy and intuitive, which is why Cypher is suitable for use by developers, professionals and users with a basic set of database knowledge.

Cypher is the official query language used by Neo4j DBMS. When using Neo4j DBMS, one can define integrity constraints by using the CREATE CONSTRAINT clause and drop them from the database by using the DROP CONSTRAINT clause. At this point of time, Neo4j enables users to define only the unique property constraint, but it only applies to nodes. This constraint is used to ensure that all nodes with the same label have a unique property value. For instance, to create a constraint that ensures that the property Name of nodes labeled Movie has a unique value, the following Cypher query must be executed:

CREATE CONSTRAINT ON (m:Movie) ASSERT m.Name IS UNIQUE

Fig. 1 shows the error message displayed to the user when the user tries to insert a movie with duplicate name, which violates the previously specified integrity constraint.

B. Gremlin

Gremlin is a graph traversal language developed by Apache Tinkerpop. Gremlin enables users to specify steps of the entire graph traversal process. When executing Gremlin query, several operations and/or functions are evaluated from left to right as a part of a chain.

At this point of time, Gremlin does not provide support for any kind of integrity constraint, which leaves a lot of space for improvement.

Node 0 already exists with label Movie and property "Name"=

Figure 1. Constraint violation error message

In the next section, it is shown how to implement support for integrity constraints in a Neo4j graph database.

III. SPECIFYING UNIQUE NODES AND RELATIONSHIPS IN GREMLIN

In the relational data model, the UNIQUE constraint is used when a column value in a table must be unique in order to prevent duplicate values to be entered into a table. The UNIQUE constraint can be specified for one or more columns in a table. For instance, if certain table columns are declared as UNIQUE, it implies that the values entered for these columns can appear only in one table row, i.e., there cannot be any rows containing the same combination of values for these columns.

It is already mentioned that in graph database theory, the UNIQUE constraint is defined as a constraint to be applied on a node/relationship property, therefore having the same meaning as the corresponding constraint in the relational database world. However, to prevent data corruption and redundancy when repeatedly inserting nodes and relationships with the same properties, we propose that the UNIQUE constraint should be and can be defined on nodes and a relationship as a whole, instead of only on some of their properties.

In [6], some implementation challenges regarding different vendors and approaches for the implementation, such as application programming interfaces (APIs), extensions and plugins, have been discussed. The research paper was concluded by choosing the API approach, so a web application has been built by using Spark, a Java web framework, and Apache Tinkerpop, a graph computing framework for graph databases, which contains classes and methods for executing Gremlin queries. The application interacts with a Neo4j graph database through the JAVA API. The purpose of the application is to showcase the usage of the unique node/relationship constraint when creating a node/relationship through executing Gremlin queries. The web application consists of a GUI where a user can create one or two nodes connected via a relationship or query-created nodes and relationships.

The UNIQUE constraint is defined in an additional graph DBMS layer, which behaves as a mediator between the graph database and the application itself. The constraint itself is implemented as a special verification method, which is called when the user wants to create unique nodes and relationships in order to check whether these nodes and relationships already exist in the database.

A. Creating one unique node

When creating one unique node, the user first needs to select a node label from the dropdown list. For instance, to create an author, one needs to select the Author label and set its property values (“firstname” and “lastname”). After that, if the Author node needs to be unique, i.e., in order to ensure that there are no nodes with the same labels and property values in the database, the UNIQUE checkbox must be checked, as shown in Fig. 2.

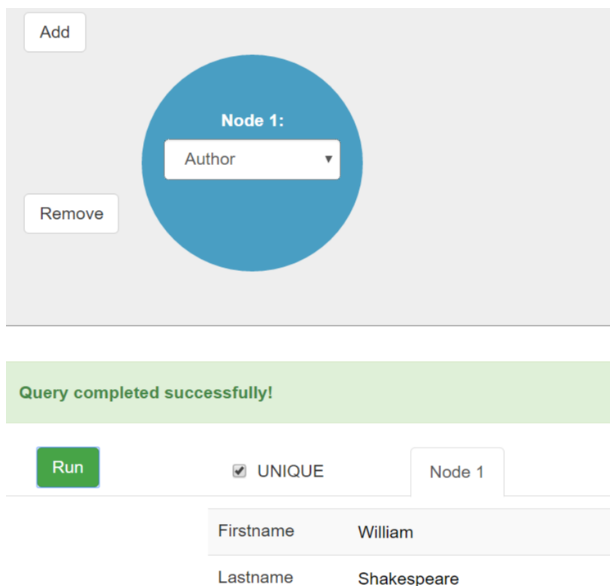

Figure 3. Creating one unique node

When running this query, the entered data is sent as parsed parameters first to the method, which source code is shown in Fig. 3, which checks if an Author node with the received parameters already exists in the database. This method retrieves all nodes (by using the *g.V()* nodes iterator) that are labeled "Author" (by using the *has()* method, which returns true if the values are equal or false if they differ) and have the same property values as the node, which was sent as a parameter to the method. If true, the method returns the existing node. However, if a node with the same label and property values does not exist in the database, it will return a NULL value. Then, a new "Author" node will be created within a Neo4j database transaction by calling the *addVertex()* method and setting the appropriate property values (Fig. 4).

```
if(g.V().has("Label", a.getLabel()).has("Firstname", a.getFirstname())
        .has("Lastname", a.getLastname()).toList().iterator().hasNext())
{
    result = g.V().has("Label", a.getLabel()).has("Firstname", a.getFirstname())
            .has("Lastname", a.getLastname()).toList().iterator().next();
}
```

Figure 2. Checking whether the entered "Author" node exists in the database

```
try (Transaction tx = db.tx()) {
    vertex = db.addVertex(a.getLabel());

    vertex.property("Label", a.getLabel());
    vertex.property("Firstname", a.getFirstname());
    vertex.property("Lastname", a.getLastname());
    tx.commit();
}
```

Figure 4. Creating new "Author" node in the database

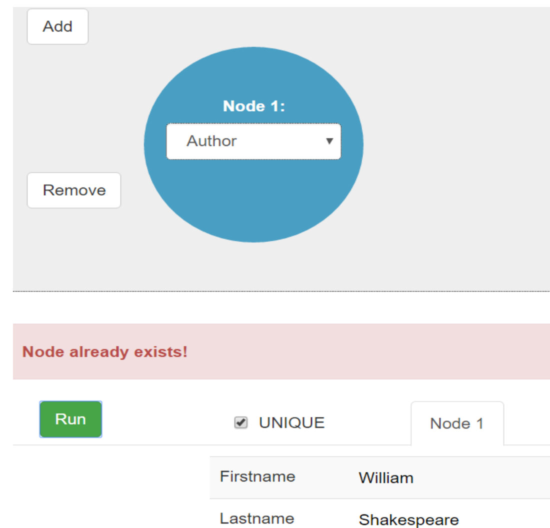

Figure 5. Error message when trying to create duplicate node

If the user tries to create another author named William Shakespeare, no changes are made to the database, i.e., the database does not change its internal state, and the result of this unsuccessful operation is a notification displayed to the user (Fig. 5).

B. Creating one unique relationship

When creating a relationship between two nodes, the user first needs to select the necessary labels from dropdown lists. For instance, if one wants to create a "BORROWED" relationship type between "User" and "Book" nodes, the aforementioned labels must be selected, and their property values defined. After having selected the required node and relationship labels and entered their property values, if the selected relationship needs to be unique, i.e., in order to ensure that there are no relationships of the same type with equal property values in the database, one needs to check the UNIQUE checkbox first (similar to the definition of a unique node).

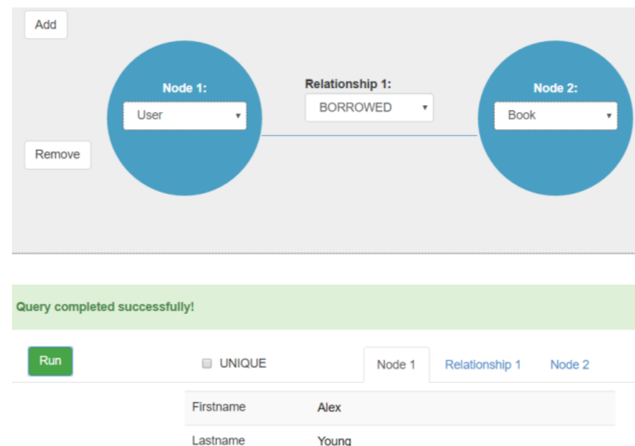

Figure 6. Creating one not unique relationship

```
try (Transaction tx = db.tx()) {
    Vertex node1 =
            Vertex.class.cast(createUser(u, createUnique).get("object"));

    Vertex node2 =
            Vertex.class.cast(createBook(b, createUnique).get("object"));

    edge = node1.addEdge(borrowed.getType(), node2);
    edge.property("DateBorrowed", borrowed.getDateBorrowed());
    tx.commit();
}
```

Figure 7. Creating new "BORROWED" relationship in the database

To show what happens if that checkbox is not checked, a “BORROWED” relationship between the user “Alex Young” and the book “Romeo and Juliet” has been created, as shown in Fig. 6. If a relationship is not specified to be unique, the Gremlin query for creating two nodes and this relationship is directly executed with the received parameters (nodes and relationship property values), which means that there is no verification for whether these objects already exist in the database (there is no call for the verification method). Each time a user runs this query, “Author” and “Book” nodes and a “BORROWED” relationship between them will be created in the database by simply calling the previously explained custom *createUser()* and *createBook()* methods. After creating the nodes, the *addEdge()* Gremlin method is called, which creates a relationship between the two created nodes and sets all necessary relationship property values through the *property()* method. The nodes and the relationship are created within a single database transaction, as shown in Fig. 7.

If a relationship is not specified to be unique, the result is duplicate data in the database, as shown in Fig. 8.

Conversely, as with creating unique nodes, if the UNIQUE checkbox when creating a relationship is checked, then the method that checks if a relationship with same type and properties exists in the database is called and executed. This method, which source code is shown in Fig. 9, performs a graph traversal in order to find the required nodes and relationship within the graph. It first retrieves all nodes labeled “User” with the property values equal to the new node that we want to create, finds the outgoing edges (relationships) labeled “BORROWED” with the same property value as the new relationship by calling the Gremlin *outE()* traversing method, and then finds the incoming vertices (nodes) of that relationship, which are labeled “Book” and have the same property values as the new node by calling the *inV()* method .

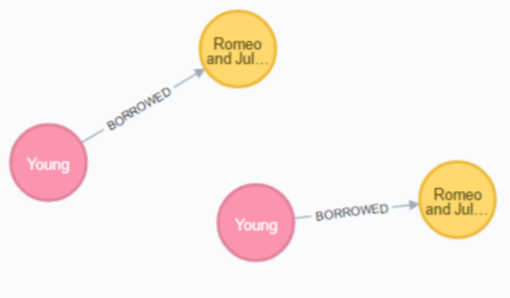

Figure 8. Duplicate relationships in the database

```
User u = User.class.cast(node1);
Book b = Book.class.cast(node2);
BORROWED borrowed = BORROWED.class.cast(rel);

if(g.V().has("Label", u.getLabel()).has("Firstname", u.getFirstname())
        .has("Lastname", u.getLastname())
        .outE().has("DateBorrowed", borrowed.getDateBorrowed())
        .inV().has("Label", b.getLabel()).has("Title", b.getTitle())
        .has("Year", b.getYear()).toList().iterator().hasNext()){
    result = true;
}
```

Figure 9. Checking whether the entered "BOROWED" relationship exists in the database

Thus, if the user tries to create a duplicate “BORROWED” relationship, which already exists in the database, then the appropriate notification message, similar to the message shown in Fig. 4, is displayed.

As already mentioned, the UNIQUE integrity constraint is implemented as a method, which is a part of the application. Its main purpose is to check whether the nodes and relationships, which the user is trying to create, already exist in the database. This is achieved by executing simple Gremlin queries that traverse the graph in order to find the subgraph corresponding to these nodes and relationships. As such, this infers that the implemented UNIQUE constraint does not affect the database performance in any way, since it is implemented through a layered approach as a method within the application. As a result, this constraint and the implemented method increase the complexity of creating nodes and relationships, and, like every other method within an application, it requires additional time to be executed (especially when performing more complex graph traversals). The Gremlin query language is, however, proven to perform well in these situations. Therefore, the cost of time necessary to execute the method is still acceptable when considering the benefits for database consistency.

IV. CONCLUSIONS AND FUTURE WORK

In this paper, the importance of integrity constraints in the database has been discussed. After giving an overview of the current support for defining integrity constraints provided by the most popular graph DBMSs, it can be said that the level of support is currently minimal and mostly theoretical, thus leaving this issue available for further research and improvements. To showcase the UNIQUE integrity constraint in graph databases, that integrity constraint was implemented as a method within an application, which performs Gremlin queries in a Neo4j database in order to check for existing nodes and relationships. Therefore, the UNIQUE constraint has been successfully implemented as a separate independent layer, which fulfills the required task (preventing duplicate nodes and relationships from being created in the database and enforcing database integrity and consistency), with minimal effect on application performance and absolutely no effect on database performance while executing queries.

In the future, this research is to be extended by implementing more complex integrity constraints, which will be discussed in our future research papers.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-513-5

REFERENCES

[1] H. Ibrahim, S. Ceri, P. Fraternali, S. Paraboschi, L. Tanca, U. S. Chakravarthy, et al., "Integrity Constraints Checking in a Distributed Database," in *Soft Computing Applications for Database Technologies*, vol. 19, no. 3, IGI Global, 1AD, pp. 153–169.

[2] H. Decker and D. Martinenghi, "Database Integrity Checking," *Database Technol.*, pp. 961–966, 2009.

[3] E. F. Codd, "Data models in database management," in *ACM SIGMOD Record*, 1980, vol. 11, pp. 112–114.

[4] DB-Engines, "Popularity ranking of database management systems," 2016. [Online]. Available: http://db-engines.com/en/ranking. [Accessed: 17-Sep-2016].

[5] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: New Opportunities for Connected Data*. " O'Reilly Media, Inc.," 2015.

[6] K. Rabuzin, M. Šestak, and M. Novak (in press), "Integrity constraints in graph databases – implementation challenges," 2016.

[7] R. Angles, "A comparison of current graph database models," in *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, 2012, pp. 171–177.

[8] "DB-Engines Ranking - popularity ranking of graph DBMS." [Online]. Available: http://db-engines.com/en/ranking/graph+dbms. [Accessed: 16-Sep-2016].

[9] Titan:db by Aurelius, "Schema and Data Modeling." [Online]. Available: http://s3.thinkaurelius.com/docs/titan/0.5.1/schema.html. [Accessed: 17-Sep-2016].

[10] OrientDB Manual, "Graph Schema." [Online]. Available: http://orientdb.com/docs/2.1/Graph-Schema.html. [Accessed: 17-Sep-2016].

[11] Franz Inc., "Introduction | AllegroGraph 6.1.1," 2016. [Online]. Available: http://franz.com/agraph/support/documentation/current/agraph-introduction.html. [Accessed: 17-Sep-2016].

[12] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Comput. Surv.*, vol. 40, no. 1, pp. 1–39, 2008.