Integrity constraints in graph databases - implementation challenges

Šestak, Martina; Rabuzin, Kornelije; Novak, Matija

Source / Izvornik: **Proceedings of Central European Conference on Information and Intelligent Systems, 2016, 2016, 23 - 30**

Conference paper / Rad u zborniku

Publication status / Verzija rada: **Published version / Objavljena verzija rada (izdavačev PDF)**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:102684

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

Integrity constraints in graph databases – implementation challenges

Martina Šestak, Kornelije Rabuzin, Matija Novak
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, Varaždin
{msestak2, kornelije.rabuzin, matija.novak}@foi.hr

Abstract. *Graph databases are becoming more and more popular as they represent a good alternative to relational databases for some problem scenarios. Searching a graph is sometimes very convenient, unlike writing complex SQL queries that require a table to be joined to itself several times. However, graph databases do not support all the constraints that are familiar and used in relational databases. In this paper, we discuss integrity constraints in graph databases and technical implementation issues that prevent these constraints from being specified.*

Keywords. Graph databases, Neo4j, Gremlin, integrity constraints, UNIQUE

1 Introduction

Graph databases are becoming more and more popular as they are constantly being developed and used in many problem scenarios [Cheng et al., 2008]. Graph databases store information in nodes and relationships between nodes. The idea of storing data in nodes and relationships is relatively new, although the graph theory is quite old. Social network analysis, recommendation systems and fraud detection represent only some applications of graph databases, and in those scenarios graph databases **outperform relational databases** [Robinson et al., 2013]. An example graph database is given in Fig. 1.

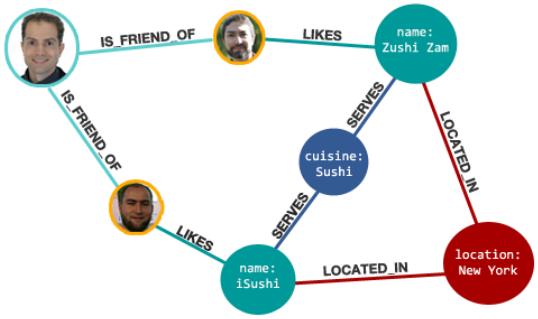


Figure 1. Restaurant recommendation graph database [Eifrem, Rathle, 2013]

One thing that is still being developed for graph databases is integrity constraint support.

Garbage In Garbage Out (GIGO) is a well-known term that describes situations in which low quality data is stored in a database, and results in the same output. Integrity constraints can be defined as general statements and rules that define the set of consistent database states, changes of states or both [Codd, 1980]. Their purpose is to ensure that data that is to be entered obeys certain rules and is valid, and that GIGO is prevented.

When talking about database constraints, we must first mention column data types, which are also important because they can prevent certain anomalies. For example, a string cannot be entered into a column whose data type is set to date or into a column whose data type is set to integer. So, proper selection of data types is important. It is also important to specify the right length for each column in order to prevent values that are too long from being stored, etc. After the right data type has been selected, integrity constraints can be specified. By using the constraints, one can restrict possible values that could be entered as column values. For example, only numbers between 1 and 100 could be entered in a field.

Once constraints are specified, the database system has to ensure that all constraints are satisfied and none are broken. Eventually, some constraints do not have to be maintained within a transaction and it is possible that some are broken, but when the transaction ends, all constraints have to be satisfied [Ibrahim, 2010].

Although constraints are specified and data is consistent, thereby satisfying the integrity requirements, this does not have to mean that data is correct. For example, let us assume that an order was delivered on the 12th of June, but somebody entered the 13th of June. Although both values are consistent and can be stored in a field whose type is set to date, one of them is not correct and is a result of a mistake. So constraints will ensure that data is consistent, but correctness cannot be ensured.

In this paper we discuss integrity constraints in graph databases. First, we describe integrity constraints and then we present which constraints are supported in current graph query languages.

2 Integrity constraints in graph databases

When talking about constraints, we distinguish:

- Column constraints
- Table constraints
- Database constraints

Column constraints are defined upon a column in a table. Examples include:

- NOT NULL: prevents a NULL from being entered into the column
- UNIQUE: ensures that a value is unique (or null, if possible)
- CHECK: ensures that the value satisfies the condition that is specified (for example, a value is between certain values, etc.). In a sense, this restricts the value of the attribute by allowing only certain values to be entered.
- PRIMARY KEY: ensures that the value is NOT NULL and UNIQUE
- REFERENCES: ensures that the value entered has to appear as a primary key value of some other (or the same) table

In some database management systems, like PostgreSQL, one can also create a domain[1] [The PostgreSQL Global Development Group, 2016]:

```
CREATE DOMAIN name [ AS ] data_type
        [ COLLATE collation ]
        [ DEFAULT expression ]
        [ constraint [ ... ] ]
Constraint can be:
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

A domain is basically a combination of a data type and a constraint. Once a domain is created, instead of a data type, a domain is specified for a column and this is sometimes very convenient.

Table constraints can be used as well because some constraints cannot be expressed as column constraints. Therefore, they are defined upon the table. For example, if a table had a compound primary key consisting of three columns, then one could not specify the PRIMARY KEY column constraint in three columns, as the PRIMARY KEY clause can appear only once within the table definition. So some constraints can only be expressed as table, and not for columns.

Triggers are very interesting as they could be used to implement more complex constraints involving more tables (database constraints) [Decker&Martinenghi, 2009]. Basically, once an event occurs (like INSERT or UPDATE), a function (procedure) is activated and several different statements can be executed as a reaction to the event.

The CREATE TRIGGER[2] statement can look different in other systems, but in PostgreSQL the syntax is [The PostgreSQL Global Development Group, 2016]:

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE |
AFTER | INSTEAD OF } { event [ OR ... ] }
        ON table_name
        [ FROM referenced_table_name ]
        [ NOT DEFERRABLE | [ DEFERRABLE ] [
INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
        [ FOR [ EACH ] { ROW | STATEMENT } ]
        [ WHEN ( condition ) ]
        EXECUTE    PROCEDURE    function_name    (
arguments )
```

Now that we know what integrity constraints are, let us take a look at how they are implemented in graph databases, and what is supported at this point in time.

Since graph databases are a relatively new category of NoSQL databases, the data consistency and integrity constraints area is still not developed in detail and provides opportunities for further improvements and studies. Some people even say that the reason for this is the flexible and evolving schema supported by graph databases, which makes integrity constraints implementation more difficult.

[Angles&Gutierrez, 2008] identified several examples of important integrity constraints in graph database models:

- Schema-instance consistency
  - Prevents incomplete or non-existent information from being inserted into the database.
  - Implies that the instance should contain only the entities and relations that were previously defined in the schema. Thus, each entity can have only those properties that were specified for that entity type or a super-type in case of inheritance.
- Data redundancy
  - Decreases the amount of redundant information stored in the database.
  - Can be solved by introducing an operation that groups entities on the basis of some of their relations, i.e., creates a unique entity for each equivalence class of duplicate entities.
- Identity integrity
  - Each node in the database is a unique real world entity and can be identified by either a

[1] https://www.postgresql.org/docs/9.6/static/sql-createdomain.html

[2] https://www.postgresql.org/docs/9.6/static/sql-createtrigger.htm

value (e.g., its ID number) or the values of its attributes.
    - Similar to primary key constraint in relational databases.
- Referential integrity
    - Requires that only existing entities in the database can be referenced.
    - Similar to foreign key constraint in relational databases.
- Functional dependencies
    - Test if an entity determines the value of another database entity.

In his other article, [Angles, 2012] considered some additional integrity constraints such as types checking, verifying uniqueness of properties or relations and graph pattern constraints.

In the relational data model, we use tables and add rows to tables. If a value for a column is not defined, NULL should be used and stored as a cell value. If a NOT NULL constraint is specified, then the value for that column has to be specified and NULL cannot be used. In graph databases, nodes and/or relationships do not have to have the same number of attributes. So, if a value is not known or defined, the attribute can be skipped. But there is also the possibility to specify a DEFAULT value for such an attribute as well, if it makes sense in a given context.

3 Graph Database Query Languages

There are two popular languages that are used for graph databases: Cypher and Gremlin.

In the next section, we will describe constraints that these languages support, though we can say that at this time the support is minimal.

3.1 Cypher

Cypher is a declarative, SQL-like query language for describing patterns in graphs using ASCII-art symbols [Neo Technology, Inc., 2016A].

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 to be used by developers, professionals and users with a basic set of database knowledge.

Cypher clauses are grouped into several categories (e.g., general clauses, reading clauses, writing clauses, etc.). The CREATE clause is used to insert data to the database. To create a new author with his own properties in a database, the following Cypher query would be executed:

```
CREATE (a:Author {Firstname: ‘Miroslav, Lastname: ‘Krleža})
```

Conversely, to retrieve all authors from that same database, the Cypher query would have the following syntax:

```
MATCH (a:Author) RETURN a
```

Cypher is the official query language of the most popular graph DBMS, Neo4j.

In Neo4j, integrity constraints are created using the CREATE CONSTRAINT clause and are dropped from a database by using the DROP CONSTRAINT clause.

Neo4j enables users to define only unique property constraints, which can be applied only to nodes. Note that the official Cypher website mentions the property existence constraint, but at the current time this constraint cannot be created in a Neo4j database.

The unique property constraint is used to ensure that all nodes with specific label have a unique value of some property. For instance, to create a constraint which ensures that the property “Name” of a node labelled Genre has a unique value, the following Cypher query must be executed:

```
CREATE CONSTRAINT ON (g:Genre) ASSERT g.Name IS UNIQUE
```

If a user tries to enter data that violates the defined integrity constraints, they will receive the corresponding error message shown in Fig. 2.

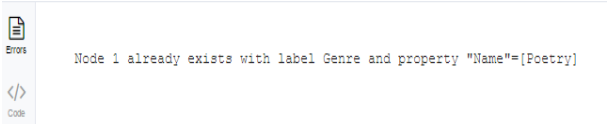


Figure 2. Constraint violation error message

3.2 Gremlin

Gremlin is a graph traversal language developed by Apache Tinkerpop. Gremlin is path-oriented, which enables it to concisely express the graph traversal process [Titan by Aurelius, 2016].

A Gremlin query is a chain of operations and/or functions evaluated from left to right. Each of these operations represents a step in the graph traversal process.

Compared to Cypher, Gremlin doesn’t provide support for any kind of integrity constraints, which leaves a lot of room for improvement. This research fits this space, as we show later on.

In the next section, we show how to support integrity constraints in graph databases.

4 Implementation issues

In this section, we describe implementation issues regarding the integrity constraint specification in graph databases. There are basically two approaches for how to implement constraints: integrated and layered. For implementation purposes we have chosen the layered approach, which means that constraints should be defined within an additional layer. In other words, we do not change the system's source code to implement constraints, which would represent an integrated approach. A layered approach has both advantages and disadvantages, but it also imposes certain implementation issues, as we show below.

To demonstrate integrity constraint implementation in a Neo4j graph database (in this case we used Neo4j Community Edition v2.3.5), a web application has been built first by using Spark, a Java lightweight web framework. After that, we had to select an approach to access the Neo4j database in order to execute Gremlin queries. This is where we encountered some issues, which we will discuss in detail later on.

Approaches to access Neo4j graph databases can be grouped into two categories:

1. Using Neo4j plugins; or
2. Using different Java APIs, drivers and native Java implementations

4.1 Neo4j plugins

This approach includes downloading an archived file from the plugin website, which contains the plugin code, configuring the Neo4j Server and deploying (registering) the downloaded plugin onto the server.
In this approach, two options were tested:

- Neo4j Gremlin plugin [Aurelius, 2016], developed and maintained by Aurelius; and
- Neo4j Server plugin, open source Neo4j distribution available on Github.

In both cases, we received error messages when trying to compile plugin code through Maven (a project management tool for Java projects), so this approach was not very effective because we did not manage to successfully connect to the database. Also, these plugins are script-based, which means they are supposed to execute Gremlin scripts on the server, which is not the functionality we were looking for.

4.2 Java APIs, drivers and native implementations

This approach requires less manual work from the developer, but it has many variations and API versions, so a significant effort needs to be made to find the best API while considering different vendors and API version compatibility.

More options were tested in this approach, which are discussed in the following subsections.

4.2.1 Tinkerpop Gremlin v2.6.0

As indicated on the official API website[3], this API represents an outdated version of the Tinkerpop framework (Tinkerpop 3 is the currently used version). Tinkerpop[4] is a graph computing framework licensed under Apache, which can be used for both graph databases and graph analytic systems.

The API can be used in both Java and Groovy implementations, but the documentation is mainly written for Groovy, so it has not been very helpful in our test case. The API was included into the project as a Maven dependency with the following syntax:

```
<dependency>
  <groupId>com.tinkerpop.gremlin</groupId>
  <artifactId>gremlin-*</artifactId>
  <version>2.6.0</version>
</dependency>
```

Unfortunately, as shown in Fig. 3, no such dependency could be found in the central Maven repository, so we gave up this option.

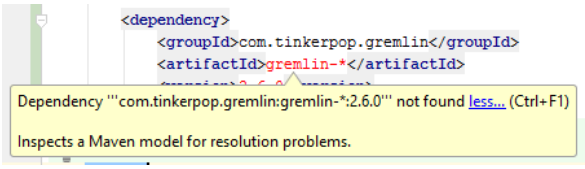


Figure 3. Error message when importing Maven dependency

4.2.2 Tinkerpop Gremlin Java implementation

Like the previous option, this API also hosts an outdated version of the Tinkerpop framework and Gremlin language documentation [Tinkerpop, 2016]. This is why it is possible to only create a TinkerGraph, which is a reference to a Blueprints graph, the generic graph API contained in the Tinkerpop2 version of the framework.

This API was included in the project as a Maven dependency using the following syntax:

```
<dependency>
        <groupId>com.tinkerpop.gremlin</groupId>
        <artifactId>gremlin-java</artifactId>
        <version>2.7.0-SNAPSHOT</version>
</dependency>
```

After successfully importing the dependency, we were able to successfully connect to the database by using the following code:

```
Graph g = TinkerGraphFactory.createTinkerGraph();
```

[3] https://github.com/tinkerpop/gremlin/wiki

[4] http://tinkerpop.apache.org/

Note that the TinkerGraph default graph structure contains some predefined vertices and edges, but it is possible to add new and custom vertices and edges.

The API provides two ways to execute Gremlin queries:

- Using the *GremlinPipeline* chaining/combinator approach, which represents a simple way of defining Gremlin-style graph traversals; and
- Directly by using two available classes (*GremlinGroovyScriptEngine* and *GremlinGroovyScriptEngineFactory*), which are useful when using the Gremlin virtual machine from within Java applications.

In our case, we tested the first, *GremlinPipeline*, approach and managed to successfully create a new vertex in the database by executing the following code snippet:

```
GremlinPipeline pipe = new GremlinPipeline();
Vertex v = db.addVertex(null);
v.setProperty("Label", u.getLabel());
v.setProperty("Firstname", u.getFirstname());
v.setProperty("Lastname", u.getLastname());

pipe.start(db.addVertex(v));
```

The result of this code is a new node with three properties, which has been added to the TinkerGraph (Fig. 4).

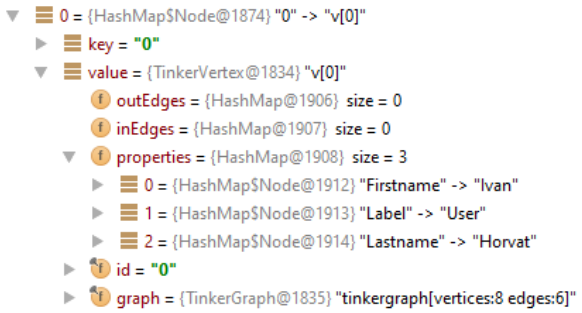


Figure 4. List of properties of the newly created node

However, even though this option has proven to be successful when it comes to connecting to the database and executing Gremlin queries, it did not allow us to create a Neo4jGraph, which is what we needed in our case. Therefore, since TinkerGraph is an in-memory graph (the database is not saved in a permanent file on a disk), when using this option, we were not able to visualize the results of the executed queries, so we also gave up this option.

4.2.3 Tinkerpop Gremlin v3.0.1-INCUBATING

The third version of Tinkerpop, graph computing framework, includes some changes compared to Tinkerpop2. Various Tinkerpop projects (Blueprints for graph model structure definition, Pipes for graph traversal, Frames for traversal, Furnace for vertex computing and Rexster as a Gremlin server) have been merged to a general term called Gremlin[5] [Tinkerpop, 2015]. Also, some syntax changes were made in terms of writing Gremlin queries, i.e., the traditional Java getters and setters have been replaced by Gremlin-Groovy syntax, which is a special Gremlin language variant.

The framework is composed of two parts:

- Components for graph structure definition, such as Graph, Element (Vertex and Edge) and Property interfaces and classes; and
- Components for traversal process definition, such as *TraversalSource* and *GraphComputer* interfaces and classes.

In this version, a new graph traversal concept called Traverser has been introduced. Traverser enables the steps in the graph traversal process to remain stateless, but it also keeps track of the entire traversal metadata.

This API was included in our project as a Maven dependency by using the following syntax:

```
<dependency>
        <groupId>org.apache.tinkerpop</groupId>
        <artifactId>gremlin-core</artifactId>
        <version>3.1.0-incubating</version>
</dependency>
```

Like in the previous option, this API supports only the in-memory TinkerGraph, which can be created by executing the following code:

```
Graph graph = TinkerGraph.open();
```

After creating the graph, the API provides methods for creating vertices and edges:

```
Vertex marko = graph.addVertex(T.label, "person",
T.id, 1, "name", "marko", "age", 29);
Vertex vadas = graph.addVertex(T.label, "person",
T.id, 2, "name", "vadas", "age", 27);
   marko.addEdge("knows",    vadas,    T.id,    7,
"weight", 0.5f);
```

When testing this option, we stumbled upon an issue when trying to create the TinkerGraph according to the official API documentation - the compiled dependency did not contain the TinkerGraph class, so it was not possible to use this API properly.

4.2.4 Tinkerpop Neo4j-Gremlin

This module[6] was developed under the Apache2 license and references only the Neo4j API without its implementation, so the implementation API needs to be added as a separate dependency [Tinkerpop, 2015]. Also, this module does not include the Gremlin Console or Gremlin Server.

[5] http://tinkerpop.apache.org/docs/3.1.0-incubating/#_tinkerpop3

[6] http://tinkerpop.apache.org/docs/3.1.0-incubating/#neo4j-gremlin

We included this module in our project as a Maven dependency by using the following syntax:

```
<dependency>
  <groupId>org.apache.tinkerpop</groupId>
  <artifactId>neo4j-gremlin</artifactId>
  <version>3.1.0-incubating</version>
</dependency>

// Neo4j implementation API
<dependency>
 <groupId>org.neo4j</groupId>
 <artifactId>neo4j-tinkerpop-api-mpl</artifactId>
 <version>0.1-2.2</version>
</dependency>
```

Unlike previously mentioned APIs, this module supports the Neo4jGraph, which can be created by calling the *open()* method implemented in the Neo4jGraph class. The method receives the Neo4j database directory path as an argument.

```
Graph graph = Neo4jGraph.open('/tmp/neo4j')
```

After successfully connecting to the database, we managed to add a vertex to the database by executing the following code snippet:

```
Vertex v = db.addVertex(u.getLabel());
v.property("Label", u.getLabel());
v.property("Firstname", u.getFirstname());
v.property("Lastname", u.getLastname());
```

One of the advantages of this option is the possibility of using the Neo4j web interface, which provides us with an overview of the current database structure, as shown in Figure 5.



Figure 5. Overview of created nodes in the database

After creating vertices and edges, the graph database can be traversed by calling the *traversal*() method contained in the graph object:

```
g = graph.traversal()
g.V().hasLabel('User').values('Firstname')
```

This option has proved to be successful in terms of both functionality (we were able to access the Neo4j database and execute Gremlin queries) and graph database visualization (we can see all changes made in the Neo4j web interface).

4.2.5 Tinkerpop Gremlin driver for Java

This driver[7] represents a reference client for Java-based applications, which enables applications to send requests to a Gremlin Server and receive results [Tinkerpop, 2015].

We included this driver in our project as a Maven dependency by using the following syntax:

```
<dependency>
  <groupId>org.apache.tinkerpop</groupId>
  <artifactId>gremlin-driver</artifactId>
  <version>3.1.0-incubating</version>
</dependency>
```

In order to connect to the database and send Gremlin queries to the Gremlin Server, it is necessary to open a new reference to localhost and create a new client, that will be responsible for sending queries/requests and receiving the results. By executing the following code, a new Client instance is created:

```
Cluster cluster = Cluster.open();
Client client = cluster.connect();
```

After executing this code, we received an error message (RuntimeException). As a result, we were not able to successfully connect to the database, so we gave up this option.

4.2.6 Neo4j Java driver

Neo4j Java driver[8] is the official driver supported by Neo4j [Neo Technology, Inc., 2016B]. The driver enables users to connect to a Neo4j graph database by using the standard Neo4j binary protocol - Bolt.

The driver was included in our project as a Maven dependency by using the following syntax:

```
<dependency>
  <groupId>org.neo4j.driver</groupId>
  <artifactId>neo4j-java-driver</artifactId>
  <version>1.0.3</version>
</dependency>
```

The connection to the database is then established by creating a new session in the Driver instance:

```
Driver driver = GraphDatabase.driver(
"bolt://localhost", AuthTokens.basic(
"neo4j", "neo4j" ) );
Session session = driver.session();
```

After creating the session, all queries were executed within that session by calling the *run*() method and passing the Cypher query as an argument:

```
   session.run("CREATE(u:User
{Firstname:'Ivan',Lastname:'Horvat'})" );
```

However, in our test case we received an error message when trying to connect to the database, as shown in Fig. 6.

[7] http://tinkerpop.apache.org/docs/3.1.0-incubating/#_connecting_via_java

[8] https://neo4j.com/developer/java/

```
Caused by: javax.net.ssl.SSLException: Unrecognized SSL message, plaintext connection?
    at sun.security.ssl.EngineInputRecord.bytesInCompletePacket(EngineInputRecord.java:1
    at sun.security.ssl.SSLEngineImpl.readNetRecord(SSLEngineImpl.java:868)
    at sun.security.ssl.SSLEngineImpl.unwrap(SSLEngineImpl.java:781)
    at javax.net.ssl.SSLEngine.unwrap(SSLEngine.java:624)
    at org.neo4j.driver.internal.connector.socket.TLSSocketChannel.unwrap(TLSSocketChann
    at org.neo4j.driver.internal.connector.socket.TLSSocketChannel.runHandshake(TLSSocke
    at org.neo4j.driver.internal.connector.socket.TLSSocketChannel.<init>(TLSSocketChann
    at org.neo4j.driver.internal.connector.socket.TLSSocketChannel.<init>(TLSSocketChann
    at org.neo4j.driver.internal.connector.socket.TLSSocketChannel.<init>(TLSSocketChann
    at org.neo4j.driver.internal.connector.socket.SocketClient$ChannelFactory.create(Soc
    at org.neo4j.driver.internal.connector.socket.SocketClient.start(SocketClient.java:7
    ... 18 more
```

Figure 6. Error message when trying to connect to Neo4j database using Gremlin driver

We were not able to connect to the Neo4j database because of some security issues (probably because we were trying to connect to an HTTPS graph database server, instead of plain HTTP). This issue required some time and effort (changing server and project configurations), but we still were unable to resolve this issue.

Also, Neo4j Java driver does not provide support for executing Gremlin queries, which is necessary for our research goals, so we gave up this option.

4.2.7 Java Core API

The Java Core API can be used in combination with Traversal API to interact with the Neo4j graph database [Haines, 2015].

Before using this API, we must include the appropriate Neo4j dependency version through Maven:

```
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j</artifactId>
  <version>2.3.5 </version>
</dependency>
```

The primary interface for that is the *GraphDatabaseService*, which contains methods for creating and querying nodes and relationships in the database.

In this case, an embedded Neo4j database was used, but creating a new database using *GraphDatabaseFactory* allows us to specify the path where all database files will be stored:

```
   GraphDatabaseService graphDB = new
GraphDatabaseFactory().newEmbeddedDatabase("p
ath/to/database/files");
```

After successfully connecting to the Neo4j database, we can call various methods for creating and querying nodes and relationships in the database. Note that all database operations are executed within a transaction:

```
try(Transaction t = db.beginTx()){
  Node node =
db.createNode(NodeController.Labels.USER);
  node.setProperty("Label", u.getLabel());
  node.setProperty("Firstname",
u.getFirstname());
  node.setProperty("Lastname",
u.getLastname());
  ResourceIterator<Node> vertices =
db.findNodes(NodeController.Labels.USER);
  while(vertices.hasNext()){
      Node user = vertices.next();
      System.err.println("\nUser: " +
user.getProperty("Firstname") + " " +
user.getProperty("Lastname") + "\n");
  }
  t.success();
}
```

By executing this code snippet it is possible to create new nodes in the database (specifically, nodes labelled User) and to retrieve all users from the database, as shown in Fig. 7.

```
User: Kristijan Horvat


User: Ivan Horvat
```

Figure 7. Result of creating two nodes using Java Core API

Since this option supports interaction with the Neo4j graph database, the results of all operations can be viewed through the Neo4j web interface.

However, even though we successfully tested this option and it met our requirements, some methods we used are deprecated (e.g. *newEmbeddedDatabase*() for creating a new database instance), and the queries are executed through the Cypher query language, so we gave up this option.

After testing all given options, we decided to use Tinkerpop's Neo4j-Gremlin API combined with its Gremlin Java implementation for our research purposes. At this time we have managed to implement the UNIQUE integrity constraint in Gremlin. Even though that constraint is currently functional, it requires some additional testing, so the details regarding integrity constraints implementation will be discussed and published in future research papers.

5 Conclusion

Integrity constraints are very important in databases as they prevent bad data from being entered and stored into a database. However, graph databases do not support all the constraints that we use in relational databases. Because of that, we examined different types of constraints in Gremlin. As it turned out, support for various constraints in graph databases was minimal.

Then, we decided to implement several new constraint types in a graph database, but implementation issues occurred, as demonstrated above.

In the end, we successfully connected to the graph database and we implemented an additional layer that supported one new constraint type that was still not supported in graph databases. In our future papers, we plan to present how we implemented the UNIQUE constraint and to implement new constraint types as well.

References

Angles, R. (2012). A Comparison of Current Graph Database Models. *Proceedings of the 28th IEEE International Conference on Data Engeneering Workshops (ICDEW)* (pp. 171-178). The Institute of Electrical and Electronics Engineers, Inc.

Angles, R., Gutierrez, C. (2008). Survey of Graph Database Models. *ACM Computing Surveys*, 40(1).

Aurelius (2016). Neo4j-gremlin-plugin. Retrieved 17.07.2016. from https://github.com/thinkaurelius/neo4j-gremlin-plugin

Codd, E.F. (1980). Data models in database management. *Proceedings of the 1980 Workshop on Data abstraction, Databases and Conceptual Modeling* (pp. 112-114). ACM Press

Cheng, J., Ke, Y., Ng, W. (2008). Efficient Query Processing on Graph Databases. *ACM Transactions on Database Systems*, 34(1), pp 1-44.

Decker, H., Martinenghi, D. (2009). *Database Integrity Checking*. IGI Global.

Eifrem, E., Rathle, P. (2013). *The most important part of Facebook Graph Search is "Graph".* Retrieved 17.7.2016. from https://neo4j.com/blog/why-the-most-important-part-of-facebook-graph-search-is-graph/

Gremlin Plugin (2015). Retrieved 17.07.2016. from https://github.com/neo4j-contrib/gremlin-plugin

Haines, S. (2015). *Programming Neo4j with Java.* Retrieved 17.7.2016. from http://www.informit.com/articles/article.aspx?p=2415371.

Ibrahim, H. (2010). *Integrity Constraints Checking in a Distributed Database*. IGI Global.

Neo Technology, Inc. (2016A). *Cypher Query Language – About Cypher*. Retrieved 17.7.2016. from https://neo4j.com/developer/cypher-query-language/#_about_cypher

Neo Technology, Inc. (2016B). *Neo4j Java driver*. Retrieved 17.07.2016. from https://neo4j.com/developer/java/.

The PostgreSQL Global Development Group (2016). *PostgreSQL 9.6beta2 Documentation*. Retrieved 17.07.2016. from https://www.postgresql.org/docs/9.6/static/index.html

Robinson, I., Webber, J., Eifrem, E. (2013). *Graph Databases*. Sebastopol, USA: O'Reilly Media.

Tinkerpop – Apache Software Foundation (2015). TinkerPop3 Documentation. Retrieved 17.7.2016. from http://tinkerpop.apache.org/docs/3.1.0-incubating/

Tinkerpop - Apache Software Foundation (2016). *Using Gremlin through Java.* Retrieved 17.7.2016. from https://github.com/tinkerpop/gremlin/wiki/Using-Gremlin-through-Java.

Titan by Aurelius (2016). Chapter 6. *Gremlin Query Language*. Retrieved 17.7.2016. from http://s3.thinkaurelius.com/docs/titan/0.5.4/gremlin.html