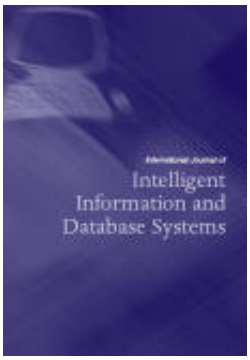
INDERSCIENCE PUBLISHERS
Linking academia, business and industry through research



International Journal of Intelligent Information and Database Systems

ISSN online: 1751-5866 - ISSN print: 1751-5858
https://www.inderscience.com/ijiids

Implementing domains in Neo4j

Maja Cerjan, Kornelije Rabuzin, Martina Šestak

DOI: 10.1504/IJIIDS.2023.10060491

Article History:

Received:	01 December 2022
Last revised:	25 August 2023
Accepted:	12 September 2023
Published online:	02 April 2024

Implementing domains in Neo4j

Maja Cerjan* and Kornelije Rabuzin

Faculty of Organization and Informatics Varaždin,
University of Zagreb,
Pavlinska 2, 42000 Varaždin, Croatia
Email: macerjan@foi.hr
Email: krabuzin@foi.hr
*Corresponding author

Martina Šestak

Faculty of Electrical Engineering and Computer Science,
Universiti of Maribor,
Koroška cesta 46, 2000 Maribor, Slovenija
Email: martina.sestak@um.si

Abstract: Data growth has led to the need to apply new ways to process data. Graph databases are increasing their use, which can be seen over the years, with the Neo4j system being the most common. The biggest problem is the small number of implemented constraints that can be used. One of the shortcomings to be explored is the need to create domains, which are used when large amounts of data are manipulated and where the value needs to be limited or when multiple attributes have the same restrictions and data types. The creation of domains can be applied multiple times. This paper summarises the implementation of domains using Neo4j and the Java programming language.

Keywords: NoSQL; domains; cypher; graph databases; Neo4j; constraints.

Reference to this paper should be made as follows: Cerjan, M., Rabuzin, K. and Šestak, M. (2024) 'Implementing domains in Neo4j', *Int. J. Intelligent Information and Database Systems*, Vol. 16, No. 3, pp.258–285.

Biographical notes: Maja Cerjan graduated from the University of Zagreb's, Faculty of Organization and Informatics with a Master's in Informatics Education. Her present position at the same faculty is that of an assistant. She is an early career researcher focused on conducting scientific research on databases.

Kornelije Rabuzin is currently a Full Professor at the Faculty of Organization and Informatics, University of Zagreb, Croatia. He holds Bachelor, Master, and PhD degrees – all in Information Science. He performs research in the area of databases, as well as in the field of data warehousing and business intelligence. He published four books and more than a hundred papers. He served as the Head of Department for Theoretical and Applied Foundations of Information Science. He has also been working as a database and business intelligence specialist.

Martina Šestak received her Master's in Information and Software Engineering from the Faculty of Organization and Informatics, University of Zagreb in 2016, and PhD in Computer Science at the Faculty of Electrical Engineering

and Computer Science in Maribor in 2022. She is currently a teaching assistant and a member of Laboratory for Information Systems at the Faculty of Electrical Engineering and Computer Science, University of Maribor. Her main research interests include graph databases, data analytics and knowledge graphs.

1 Introduction

An integral part of any organisation is a database, where data is structured in a way that meets the needs of users. The amount of data that IT systems, in the majority of organisations, need to process is increasing every day. The requirement for application of new and better solutions for which relational databases do not have satisfactory characteristics is needed. In 1998, a new database called 'Strozzi NoSQL' was created by Carlo Strozzi, where new ways of storing, accessing data and performing data manipulation were presented for the first time. Data were retrieved using shell scripts, while tables were stored in an ASCII file (Domdouzis et al., 2021). A non-relational database is a relatively newer term that uses a flexible model that can process large amounts of data and perform operations on more complex data structures than relational databases [Stojanović, (2016), pp.44–47]. Non-relational databases do not have a strictly defined schema, and different types of data can be processed. The ability to process large amounts of data is highlighted, meaning that systems can perform more tasks with concurrent data processing. This represents just some of the advantages of NoSQL databases [Sambolek, (2015), p.3]. These databases can be divided into key-value databases, document databases, column stores and graph databases. In this paper, the focus has been exclusively on graph databases.

A graph database can be defined as a database solution where data is manipulated through operations, such as create, update, delete and read, but carried out on a graph structure. The difference between relational and non-relational graph databases is visible in how data is stored, i.e., the relational data must be stored in structured tables and strictly follow the defined database schema. Graph databases have a flexible model, where data is stored as nodes and edges of a graph and thus can be changed according to the needs of the flexible database schema (Domdouzis et al., 2021).

An integral part of graph databases is a graph showing real-world entities' relationships. Hence, graphs are used in highly connected application domains, which we can represent as graphs with nodes and edges (Rohit, 2015). Each node is an entity representing some of the domain data and has a label by which it can be inferred whether a node represents a person, object or something else. Furthermore, two nodes are connected by an edge that represents a relationship between a pair of entities. The graph database integrity is, among others, maintained by integrity constraint rules, which ensure the accuracy and consistency of data stored in the graph database.

Currently, graph databases only support the uniqueness, existence, and constraints placed on node keys. Intuitively, this list should be extended to meet the maturity level of relational databases. For this reason, we propose a new integrity constraint called domain constraints for graph databases. When looking at relational databases, domains are available in several systems (e.g., PostgreSQL). Domains are beneficial, especially when working with large amounts of data, where the same attributes with the same constraints

and data type can be found in numerous tables. Creating a domain is one of the best solutions to avoid data redundancy. The name of each domain is defined, which will be used when a data type is added to a new attribute that is entered.

As already mentioned, we propose a domain constraint to be included in the graph database schema. We present the definition of such a constraint in the context of the graph database schema and present the steps required to create a new domain constraint rule within a graph database. We demonstrate the domain definition and usage process by setting up a sample exams graph database implemented in the Neo4j Graph Database Management System (GDBMS). A domain is created in the database, which is a shape that contains the constraints needed to enter the value of ECTS points properly. The following restrictions have been implemented within this domain:

- A restriction that prevents the entry of values less than one or greater than eight.
- The ability to enter only one number.
- A constraint that allows the user to enter only an integer as a data type.

In this way, the user can directly define multiple domain constraints within the Neo4j GDBMS. As an additional integrity mechanism, we implemented a trigger that checks each transaction before entering the data into the database to prevent any incorrect values from being stored in the database. A Neosemantics/(n10s) plugin with awesome procedures on cypher (APOC) triggers is used to implement the domain constraint. The commands are written in the new shapes constraint language (SHACL) syntax, which is included in the cypher graph query language. As an alternative to database triggers and SHACL queries, we created a set of Java-based methods for creating and evaluating Neo4j domain constraints during insertions.

The next chapter will present some related papers that have emerged over the years on topics related to graph database integrity. This is followed by more details about graph databases, the query languages used to work with them, the implementation of domain constraints and the validation of the proposed approach on a selected use case. Finally, the possibilities of conducting further research on the topic and conclusion are described.

2 Related work

When it comes to domains in graph databases, it must be taken into account that there is not much available literature since graph database integrity has yet to be studied in detail thus far. Although there is limited literature on domains in the context of graph database integrity constraints, many authors studied related topics on graph databases, which we will mention in the remainder of this section.

- ‘Survey of graph database models (2008)’ – the paper was made as a survey on graph database models. In this survey, the authors discuss the suitable use cases for using specific graph database models. A comparison was made with other database models (relational, network, object-oriented, semantic, etc.). The paper also presents an overview of available integrity constraints supported by the identified models, such as integrity constraints, referential integrity, identity constraints, etc.
- ‘Graph database applications and concepts with Neo4j (2013)’ – the paper compares relational and graph databases using systems that support them: Oracle, MySQL and

Neo4j. The authors investigated whether graph databases can completely replace relational databases. The conclusion is that it is necessary to consider each system's needs and requirements and decide which database type is most suitable for use. However, traditional relational databases have the best characteristics for the most common uses. Both types have their advantages and disadvantages, and the emphasis in the paper was placed on various features of the data model, queries and data structure.

- 'Querying graph databases (2013)' – the authors of this paper studied the problems related to queries that are performed in graph databases, where most attention was paid to path queries and extensions with inversions and conjunctions. Nevertheless, limitations and optimisation of queries in the presence of these limitations were also observed. The research continued in the direction of expanding the languages used in working with trails, where trails are treated in different ways. A comparison of the semantics of systems working with different languages was made, so the semantics of paths could be clarified. There are basically two semantics of paths. The first is based on some simply defined paths that lead to unsolvability when talking about the complexity of data, and the second is based on some arbitrary paths.
- 'Graph database – an overview (2014)' – a detailed overview of graph databases was made throughout this paper. One can see what graph databases are, what graphs are and what properties they consist of, and the information on their performance, flexibility and agility. The authors present real-world use cases suitable for graph databases such as social networks, telecommunications, security, bioinformatics, etc. Finally, with the help of graphical representation, a comparison was made between the most famous relational and graph databases.
- 'Graph database: a survey (2015)' – in this paper, an overview of different types of GDBMSs was made, such as Neo4j, Dex, Infinite Graph, Infogrid, HyperGraph, Trinity and Titan. Each listed GDBMS has its features, structures, models, APIs, and protocols. In this paper, a comparison of GDBMS features, their models and applications was made. As a result, the authors present a summary of query languages used in each graph database, their availability and usability and available features.
- 'Graph databases – are they really so new (2016)' – this paper provides an overview of different types of databases that store and manipulate data similarly, namely hierarchical databases. A detailed insight into network databases and an example of working with network databases are also presented. The paper also includes an analysis of graph databases, how to work with graphs, and examples of creating nodes and connections between nodes. In the end, a comparison of network and graph databases was made because they include similar concepts of nodes and edges (Maleković et al., 2016).
- 'Integrity constraints in graph database implementation challenges (2016)' – the paper describes integrity constraints present both in general and in graph databases, where the first part defines and explains constraints on columns, tables and databases. Examples in PostgreSQL are also given, as well as an explanation of integrity constraints in graph databases and examples in Cypher. The second part of the paper presents challenges that arise during the implementation of integrity

constraints with respect to the specifications. The implementation was done in Neo4j GDBMS by using the layered approach.

- 'Conceptual and database modelling of graph databases (2016)' – the first part of this paper includes a brief introduction to graph databases, graphs and graph database schemas while observing the limitations of integrity constraints. The second part of the paper covers the problem of the non-existence of conceptual modelling, where the authors propose an approach to transforming the conceptual schema of a graph into a graph database schema. The consistency of the graph database was also studied since integrity constraints were not explicitly identified and set as they should be. The relationship between conceptual model schemas and database models was also analysed.
- 'Implementing check integrity constraint in a graph database (2016)' – one part of the paper contains an insight into graph databases, which includes both the constraints and the level of support needed to determine the constraints in the Gremlin and cypher graph query languages. The authors explained the implementation of the new check integrity constraint in the Neo4j GDBMS. Examples of the implementation of the author and book nodes are given, as well as an overview of how the entered values were checked, e.g., when checking whether the user entered books that were published in a certain period, etc. (Rabuzin et al., 2016a).
- 'Implementing unique integrity constraint in a graph database (2016)' – as the extension to the previous paper, the authors continued implementing the uniqueness integrity constraint. The paper began with a brief insight into the existing integrity constraints in graph databases, and the research done so far on the topic of integrity constraints. Also, the authors wanted to show the level of support for the constraints provided by some of the known GDBMSs and finally presented the implementation of the uniqueness (UNIQUE) constraint. A sample database was prepared in the Neo4j GDBMS, and the UNIQUE integrity constraints were implemented by using the Java programming language (Rabuzin et al., 2016c).
- 'Integrity constraints in graph databases (2017)' – the Neo4j GDBMS was used to study the resulting database schema, along with integrity constraints still in development for this paper. As the essential part of this paper, an extension was made within Neo4j and the query language that this system uses (cypher). The extension includes a complete syntax for implementing integrity constraints.
- 'Creating triggers with trigger by example in graph databases (2019)' – the paper describes how to design and implement triggers in graph databases. For this study, a graph database and a graphical user interface (GUI) were created, which were used to create triggers stored as event-condition-action (ECA) rules. An overview of the work made on topics related to graph databases was made, and the trigger-by-example approach was explained and presented, ending with detailed explanations to explain graph databases and approaches related to trigger-by-example (Rabuzin and Šestak, 2019).
- 'A review on graph database and its representation (2019)'– in this paper, the authors wanted to present the use of graph databases with a real scenario, using professors, subjects and 'research scholar' relationships as examples. Furthermore, different

graph database models (property graphs, hypergraphs, triple stores) were presented and explained in detail in the same example. At the very end, relational databases were compared with graph databases.

- 'Defining referential integrity constraints in graph-oriented datastores (2020)' – this paper presents the approach and solution used in defining referential integrity constraints. To specify the referential integrity constraint (RIC), creating a domain-specific language containing clauses for configuring the RIC definition (cardinality, additional conditions, bidirectionality, etc.) was necessary. The implementation was successfully performed with the help of model-driven engineering (MDE) techniques.

Only selected papers that are thought to be the most relevant to our topic were chosen and presented in chronological order. The emphasis was on discovering works that discovered approaches and processes for implementing different constraints already available in relational databases but missing from non-relational databases. This was motivated by a limited number of integrity constraints currently supported by the Neo4j GDBMS, for which it was found that it only supports the NOT NULL and UNIQUE constraints at the moment. While certain constraints have been studied and put into practice, the majority of constraints have only been proposed theoretically, with no clear indication of their implementation details.

3 Graph databases

Although graph databases found their true application only a few years ago, graph theory is a topic that has been worked on for hundreds of years, but not under that term. The problem of the Seven Bridges of Koeningsberg was modelled with the help of graph theory, where Euler tried to come up with a solution to cross the seven bridges through four parts of the city only once. Euler modelled a graph with the same number of peaks as cities and several edges equal to the number of bridges between those cities. Although various historical problems have been solved by graph theory, the issue has not been explored so profoundly (Fošner and Kramberger, 2009).

From the above example, it can be concluded that graph databases store domain-related data in the form of graphs, which have their foundations from other exact sciences. While relational databases use tables, here, one uses nodes and edges. It can be said that graphs are based on the mathematical graph theory, where, as has been defined before, each graph should contain (Olivera, 2019):

1 Vertices or nodes representing the entities of the domain being observed.

2 Properties that represent the basic information about nodes, being very similar to the properties of tables in relational databases.

3 Edges/relations/arcs that define the relationship between two nodes.

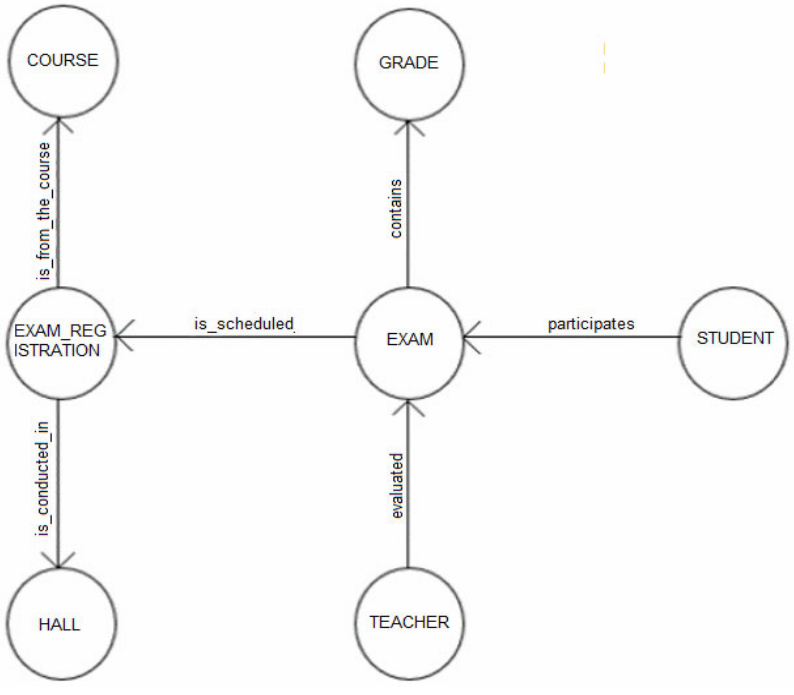
An integral part of any graph database is a graph, i.e., the data structure that has been defined as an ordered pair of nodes and edges, which can be represented as G = (V, E), where V is a set of graph nodes, and E is a set of edges. Edges can be directed or

undirected, while data manipulations are performed using graph operations (Chen et al., 2020).

In relational databases, the referential integrity constraint helps to achieve good connectivity between tables, protecting data from improper handling (deletion or modification). It can be said that referential integrity is crucial for the implementation of a relational database to be valid and to meet the rules at all levels. In graph databases, some rules must be followed to properly create the graph. When creating graphs, we need to consider the nodes that will be created, node types and the connections that will be defined between these nodes. Every edge should have its own start and end node, and deleting a given node would not be allowed without deleting all edges related to that node (Olivera, 2019).

A graph database can be considered as an instance of its schema containing a graph representation of the data. Graphs are needed to understand how to interpret all data stored in a graph database. In this implementation, first and foremost, the given constraints have to be respected, and some of them, as mentioned above, are referential integrity, identity constraint, functions, and dependency on inclusion. Graph databases can handle large amounts of data since the database queries are executed on subgraphs (a subset of the entire graph), so the characteristics of the database will not change no matter how much data is stored in the database (Domdouzis et al., 2021).

Figure 1 Simple graph display



The wide application of graphs can be seen through two general examples. The first example is one of the most famous social networks, Instagram or Twitter, where one can define a graph by putting users in nodes and defining the connections between them depending on whether users follow each other. Therefore, there can be only one

connection between two nodes if one user follows the other or two connections if both users follow each other. Such graphs are applied in different organisations to meet all requirements, achieve the desired goals, and are created using data modelling tools (Chen et al., 2020).

Figure 1 depicts a simple graph where it can be seen the types of nodes that were used to create the database in our practical example discussed later. There are seven types of nodes, and each type has several node instances with their own properties. Between each node, a named connection that connects the nodes into a meaningful whole can be seen.

4 Graph database query languages

So far, it has been shown and explained what graphs look like as an integral part of graph databases, but what is also interesting is how to create and manipulate data using query languages. In relational databases, the most commonly used query language that is used to manage data is the structured query language (SQL). In graph databases, it is possible to use two languages for writing queries:

1 Gremlin, which is a low-level language and is used in various programming languages.

2 Cypher, which can be declared as the most commonly used open-source language, being also an excellent tool to learn how to operate graphs.

Each of these languages will be briefly presented in the following subsections according to their basic characteristics, query syntax, etc. We also present the syntax of the SHACL, a query language used to check and describe RDF graphs according to a set of conditions.

4.1 Gremlin

Gremlin is a low-level language used for graph traversals with compact syntax, created under the auspices of the Apache ThinkerPop framework. It can be declarative or imperative (Robinson et al., 2015), and it does not support any integrity constraints or provide a way to extend the graph traversal process beyond the scope of the work. However, integrity constraints can be made, as Maleković et al. (2016) have shown in their research. The authors state that, unlike Cypher, Gremlin can perform complex queries since the whole process of execution is divided into a certain chain of operations, where the results do not need additional value conversion (Maleković et al., 2016).

This can be used for graph queries in various programming languages, and to better illustrate the Gremlin syntax, examples of queries for creating nodes, connections between nodes, and retrieving data will be presented.

For instance, the addV (add Vertex) command is used to add nodes:

```
g.addV (‘college’). property (‘id’, 1).property (‘title’, ‘Databases’). property (‘ects’, 6)
```

To add connections between nodes, the addE (add Edge) command can be used:

```
gV (). hasLabel (‘course’). has (‘title’, ‘Databases’). addE (‘something’). to (gV (). hasLabel (‘ course’). has (‘title’, ‘Database basics’))
```

The following syntax is used to retrieve data:

```
g.V (). hasLabel ('course'). has ('title', 'Databases').
```

4.2 Cypher

Cypher can be defined as the declarative and most commonly used query language for graphs in Neo4j and graph databases in general. It allows users to search, store and manipulate data using available create, read, update and delete (CRUD) operations and test and update graphs. Also, by using Cypher, it is possible to describe the visual patterns found in a graph, as well as to create and define nodes, edges, and properties, and use a simple SQL-like syntax. ASCII syntax is used to describe the forms, so writing and reading queries should be fine if one has experience in standard querying.

In addition to manipulating data, this language offers the ability to use different ways of filtering, grouping and retrieving only certain data that is needed at the time, which is extremely important when there is a selective display of data. Various aggregate functions can also be used to perform data aggregations. The new version provides the ability to work with dates and calculate the duration. As a good feature of the system that can be mentioned, there is an openCypher initiative, where users can contribute to the development of the language by correcting errors observed during operation (Neo4j, 2021a).

An example of creating nodes in Neo4j using Cypher is the following:

```
CREATE (Smith: Student {student_id: 1, surname: 'Smith', name: 'Christian', oib:
34214356765, year_study: 2})

CREATE (Exam1_mathematics: Exam {course: 'Mathematics', deadline:
localdatetime ('20200630T11: 00: 00'), student_id: 1, application_date: localdatetime
('20200620T09: 00: 00'), grade: 1, exam_name: Exam1_ mathematics '})
```

In the above example, two nodes are created. One of type *Exam* called *Exam1_mathematics*, and the other of type *Student*, having used the name of the node Smith. The way to write the CREATE command is as follows:

First, the command's name is specified, then the node's name, the node's type, and the properties of the newly created node are defined within the curly brackets.

Figure 2 Display of the connection between the nodes (see online version for colours)

Each node has properties that describe its so-called descriptors. Once the two nodes are created, an edge can be established between them. The name of the edge between the *Smith* nodes of the Student type and *Exam1_mathematics* of the *Exam* type is PARTICIPATES.

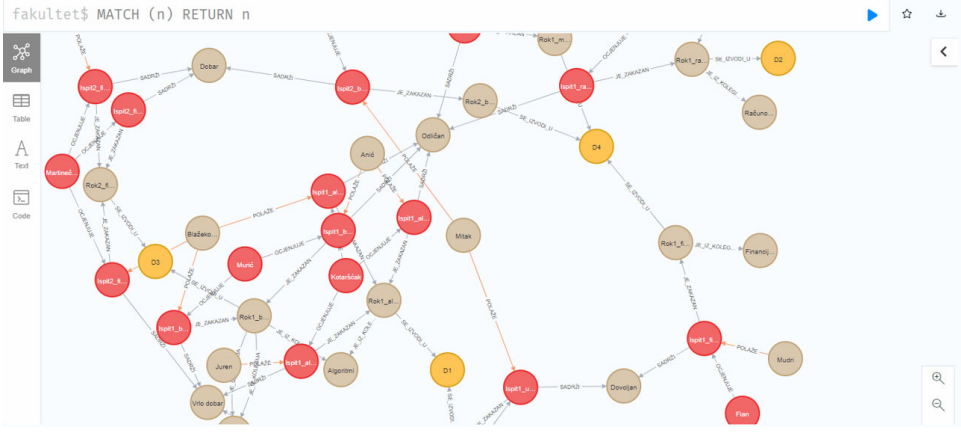
```
MATCH(Smith:Student), (Exam1_mathematics: Exam)
WHERE Smith.student_id = 1 AND Exam1_mathematics.student_id = 1
CREATE (Smith) – [r: PARTICIPATES] -> (Exam1_mathematics)
RETURN Smith, Exam1_mathematics
```

The merge via a property called *student_id* was created. The nodes are entered first, and then the *student_id* is checked. When an exam contains the same *student_id* as in the *Smith* node, an edge is created.

If we want to retrieve all nodes from the database, the following command could be used:

```
MATCH (n) RETURN n
```

Figure 3 Display of all nodes in database (see online version for colours)



4.3 SHACL

As mentioned, SHACL is a query language used to check and describe RDF graphs according to conditions. SHACL form graphs work by validating an existing graph against the conditions given and ultimately executing them as part of a commit() transaction. To avoid problems, the best is to execute transactions one after another since validation may not be performed correctly when two transactions are performed at once. In these cases, data in the database may be validated, newly created forms updated, etc. If some already created forms are updated, the transaction will not be executed if the data entered is not correctly entered.

For this reason, validation is performed again to see if everything is written correctly. If so, the commit will confirm the changes and enter the updated data in the database. Writing commands and creating checks is done using features (*sh: targetClass, sh: targetNode, sh: path*, etc.). A number of features can be used, but it has to be kept in mind that many of them are not implemented. In the following example, it can be seen how validation is done as the most important part of the SHACL language. If there are nodes with properties that belong to them, it can use SHACL to check if the property

belongs to the requested node. The following examples can show how validity works (Rdf4j, 2021).

```
ex: CourseShape
a sh: NodeShape;
sh: targetClass ex: College;
sh: property [
sh: path ex: ects;
sh: datatype xsd: integer;
]
```

A *college*-type node and the corresponding object ECTS are considered in the proposed example. To get some initial data, first, the commands are entered under number 1, and to save the data in the database with the help of a transaction, there is the need to enter what is below number 2. What is in the background writing these the command actually serves to check whether the *ex: ects* is of type *ex: Course* and whether it may already be stored in the database as *ex: Course*.

```
sh: targetClass ex: College
sh: path ex: ects '1'
```

4.4 Shapes

To verify and enforce data restrictions inside the graph, SHACL can be implemented in Neo4j, a well-known graph database. SHACL shapes are node structures and property restrictions that the graph's data must follow. With nodes denoting resources and relationships denoting connections between them, the property graph model of Neo4j and SHACL are well matched. When implementing SHACL in Neo4j, users always start by creating a collection of SHACL shapes using the SHACL expressions and defining the node types, property requirements, cardinality restrictions, and validation requirements. These shapes may be saved as specialised nodes in the graph or as SHACL-specific nodes or characteristics.

The SHACL documentation distinguishes between two categories of shapes:

- Node shape – denotes a shape that is not a component of the triple with the predicate sh:paths. The node shape should be specified as a SHACL instance of *sh:NodeShape*. They do not place as much emphasis on the node's property values, in contrast to property shapes, which do.
- Property shape – represents a form in the shapes graph that is a component of a triple with the predicate *sh:path*. It is advised to declare the property shape as a SHACL instance of the *sh:PropertyShape* class.

The constraint component parameters allow each shape to declare constraints. For instance, the *sh:MinCount* parameter is declared by the *sh:MinCountConstraint*

Component component to signify the need for a node to have a minimal quantity of values for a particular attribute.

Specific components specify only one parameter. As an illustration, *sh:ClassConstraintComponent* only has one parameter – *sh:class*. Each value of these parameters is defined as a separate constraint, and they can be utilised more than once in a single shape. Such a statement is interpreted as a conjunction, meaning that all limitations are observed (W3C, 2017). All examples are available on W3C official site listed in the literature.

```
ex:TestShape
    a sh:NodeShape ;
    sh:property [
        sh:path ex:customer ;
        sh:class ex:Customer ;
        sh:class ex:Person ;
    ].
```

According to the example, the *ex:customer* property values must be SHACL instances of both *ex:Customer* and *ex:Person*.

The following steps can be taken to define constraints using various property shapes:

```
ex:MultiplePatternsShape
    a sh:NodeShape ;
    sh:property [
        sh:path ex:name ;
        sh:pattern '^Start' ;
        sh:flags 'i' ;
    ];
    sh:property [
        sh:path ex:name ;
        sh:pattern 'End$' ;
    ].
```

For each form, a severity parameter can also be defined. Violation is the present value for severity.

Three levels of severity exist:

- Info – non-critical and shows an information message.
- Caution – non-critical and shows a warning.
- Infraction – imperative.

By setting *sh:deactivated* to true or false, every shape can be made inactive.

The following list presents all constraint components that can be set in SHACL shapes:

- Non-validating property shape characteristics
  1 sh:name, sh:description, sh:order, sh:group and sh:defaultValue
- Value type constraint components
  1 sh:class, sh:datatype and sh:nodeKind
- Cardinality constraint components
  1 sh:minCount and sh:maxCount
- Value range constraint components
  1 sh:minExclusive, sh:minInclusive, sh:maxExclusive and sh:maxInclusive
- String-based constraint components
  1 sh:minLength, sh:maxLength, sh:pattern, sh:languageIn and sh:uniqueLang
- Property pair constraint components
  1 sh:equals, sh:disjoint, sh:lessThan and sh:lessThanOrEquals
- Logical constraint components
  1 sh:not, sh:and, sh:or and sh:xone
- Shape-based constraint components
  1 sh:node, sh:property, sh:qualifiedValueShape, sh:qualifiedMinCount and sh:qualifiedMaxCount
- Other constraint components.
  1 sh:closed, sh:ignoredProperties, sh:hasValue and sh:in

In Neo4j, every option is easily accessible. It is important to remember that components of the 'property pair constraint' and 'other constraint components' may require some external tools or custom processes for complete implementation.

5 Domains

Several limitations can be observed looking at relational databases and different database management systems (DBMSs) that seek to preserve data integrity. Restrictions can be placed over the entire database or over only certain parts such as tables or columns. Each constraint can be seen as a criterion that must be met for data entry into the database to succeed. For instance, if the defined rules are not followed, enrolment in the Database course will not be possible. Some of the existing and very well-known limitations that are constantly encountered in general are (Carić and Buntić, 2015):

- Primary key – a unique identifier of each record, where the primary key can have one or more attributes that have unique values.
- Foreign key – needed to be able to connect tables in the right way.

- Not null or null – to determine whether the data is required for entry or not.
- Unique – a constraint that determines the uniqueness of the data, which means that it will not be possible for an attribute to have a repeated value.
- Check/between – implies a range of values that an attribute may have, e.g., ECTS between numbers 1 and 8.
- Default – the value will be assumed an initial value by applying this constraint.
- References – used to merge tables where the name of the table and the attribute are specified.

On the other hand, there are GDBMSs such as Neo4j where only some limitations are available. When creating a database, nodes must exist. Each node must have its label (name) that must be unique in order to be able to distinguish them. Each node has assigned properties and values that belong to it. When nodes are created, connections must be established between them. A node cannot be created without any properties or if the property value variants are not unique. Deleting the required properties will automatically result in an error. All the limitations that must be adhered to when working with graph databases will now be explained in more detail. The limitations currently available in Neo4j are as follows (Neo4j, 2021b):

- Node unique property constraint – nodes can have unique tag property values, and if there is a need for multiple property values in a single node to which this constraint should be placed, then this combination of values must also be unique.
- Restriction on the existence of node properties – each node must have properties with values that belong to a particular label. When we want to create a new node for a label and omit the property, it will report an error. Queries that are used to create new nodes and some specific labels without entering the type of a property will also fail and report an error.
- Restriction on the existence of a property connection - this restriction serves to always merge nodes over the same properties.
- Node (key) limit – the key is placed in each label with certain properties. The values contained within this property must be unique.

One of the most interesting limitations is the creation of domains that are still unavailable in graph databases. To begin with, the creation of domains in PostgreSQL is explained (PostgreSQL, 2021) as one of the relational DBMSs that can create them. This paper aims to implement a similar way of creating domains in graph databases in the Neo4j system. In the PostgreSQL system, there is the possibility of creating new domains by using the CREATE DOMAIN command, which is ultimately a property, and in accordance with the SQL standard. As a result of creating a domain, a new type of data is generated, which will already contain some built-in restrictions that will have to be respected when entering data. Each domain that is created must have its unique name among the variants that are within its scheme. Each domain name must be assigned to a specific type of data that is required. In the example of this paper, when talking about ECTS points assigned to each course, it is necessary to have an integer data type, so the domain would be called ECTS and would be displayed as a numeric value. For example,

suppose one can have a personal identification number (PIN) in several different tables instead of defining a constraint in each table. In that case, one can create a domain that will have a predefined constraint within it. So, in detail, when assigning a data type, a PIN domain will be included as a type and skip writing restrictions. This is very handy in databases with large amounts of data since attributes with the same types and constraints are repeated in multiple tables.

Restrictions that are required must be placed after the name and type of data. Some of the restrictions that can be used are NOT NULL, NULL and CHECK. In this case, a CHECK is needed since it is the objective to check if the user entered values in the ECTS field are between 1 and 8, and for this reason, it is placed BETWEEN 1 AND 8. The known constraint must have its name, and if not defined, the system will automatically assign a name. After creating a restricted domain, a table must be created in which that domain is used.

```
CREATE DOMAIN name [AS] data_type
[COLLATE collation]
[DEFAULT expression]
[constraint [...]]
```

Where are the restrictions:

```
[CONSTRAINT constraint_name]
(NOT NULL | NULL | CHECK (expression)}
```

In this case, the following constraint in PostgreSQL would be written.

```
CREATE DOMAIN type_ects AS INTEGER
CHECK (VALUE BETWEEN 1 AND 8)
```

After the domain is created, the course table is also created.

```
CREATE TABLE COURSES (
id integer primary key not null unique,
Name text NOT NULL,
ect type_ects NOT NULL
);
```

The implementation of the Neo4j extension will be presented below.

6 Domain constraint

Working with databases, database practitioners quickly realise that constraints are significant for creating a good and functional database. Many systems have various limitations that can be applied, but Neo4j is not one of them. The only available

constraints that can be currently used are uniqueness, node key constraints, and exist. Something that could certainly be listed as a weakness of this system is that too few limitations are implemented in such a large and developed system. For this reason, various solutions have been sought regarding how to set a limit on a property in order to limit inputs.

We propose to create a domain constraint defined as a SHACL shape with the following mandatory properties:

- Shape name – unique string under which the constraint will be saved in the database (e.g., CourseShape, PersonShape, AccountShape).
- Node label – referring to the node label in the database on which the constraint is imposed, (e.g., nodes of type course, person, account).
- Property name – denotes the property name of a given node type which value requires checking, (e.g., *course ects*, *person social security number* (*SSN*), *bank account balance or account type*).
- Property constraint – defines restrictions on the value of a given property in the form <constraint type>:<allowed values>. Some possible property constraint types are:
  a *Sh:has value* – property value must be among the allowed values (e.g., course name must be databases, account type must be savings).
  b *Sh:in* – property value must be a member of a pre-defined array of allowed values (e.g., course name must be databases or programming, person gender can be male, female or other).
  c *Sh:pattern* – property value must match a given regular expression (e.g., course name must start with 'databases', person SSN must include precisely ten digits).
- Minimum and maximum cardinality – define if the property is mandatory or optional (for the *sh:minCount* constraint, this can be 0, 1 or more) and the maximum number of values entered for a given property (1 or more), (e.g., *person SSN* must have *sh:minCount* and *sh:maxCount* both set to 1, as it is a mandatory attribute and a person is allowed to have only one SSN).
- Data type – restricts the allowed data type for property values to an instance of RDF data types, (e.g., *xsd:string* for account type or person gender, *xsd:integer* for person SSN). This could be extended to multiple value types allowed by using the *sh:datatypeIn* SHACL constraint.

Therefore, the general syntax representing the domain constraint rule defined in SHACL can be summarised as follows:

neo4j: <shape_name> a sh: NodeShape;

sh: targetClass neo4j: <node_label>;

sh: property [

sh: path neo4j: <property_name>;

sh:<property_constraint>;

sh: minCount <min_cardinality>;

```
sh: maxCount <max_cardinality>;
sh: datatype xsd:<data_type>;
]
```

Based on this definition, it can be observed that the definition setting restrictions on multiple properties of the same type of node simultaneously. In general, we can specify the allowed data types, cardinality and various kinds of values allowed for a given property. In addition to defining specific values and value ranges for property values with *sh:hasValue*, *sh:in* and *sh:pattern* SHACL operators, it is also possible to combine such value patterns with *sh:and* and *sh:or* operators to gain a more detailed specification of the allowed values. For instance, we could create a domain constraint for nodes labelled *Person*, which sets a person's firstname property to be mandatory, unique and a string, and the SSN property a ten-digit mandatory and unique number, we would specify a domain constraint as a combination of several restrictions with the following SHACL syntax:

```
neo4j: PersonShape a sh: NodeShape;
sh: targetClass neo4j: Person;
sh: and (
    [sh: property [
        sh: path neo4j: firstname;
        sh: minCount 1;
        sh: maxCount 1;
        sh: datatype xsd:string;
        ]
    ]
    [sh: property [
        sh: path neo4j: ssn;
        sh: minCount 1;
        sh: maxCount 1;
        sh: pattern '^ \d{10} $';
        sh: datatype xsd:integer;
        ]
    ]
).
```

Our current implementation of SHACL-based domain constraints enables users to create restrictions on node properties. These restrictions correspond to the fundamental

definition of domain constraints (property type, minimum and maximum cardinality, and the range of acceptable property values), as outlined in Section 5. Thus, our proposed approach can be employed in a GDBMS environment, which supports a semantics plugin able to run SHACL queries (e.g., Neosemantics in Neo4j). Then, the user can use the SHACL syntax presented above to create a domain constraint on nodes in the database. Once created, the constraint rule will be evaluated during the insertions or updates of node properties specified by the constraint. The list of current restrictions will be extended to include other restrictions supported by the SHACL syntax in our future implementations. Furthermore, the SHACL domain definition syntax does not incorporate any mechanisms to verify the accuracy of constraint properties, given that the underlying W3C standard does not support recursion or other validation methods. This leads to the potential for users to define domains in which allowed property values could potentially map to empty sets (for instance, allowed values specified as an intersection between two separate ranges). This issue could be addressed by potentially introducing an additional syntax validation step during the domain creation process in Neo4j.

As the next step, to implement domain constraints in Neo4j, we propose to create restrictions using APOC triggers and the Neosemantics (n10s) plugin with SHACL check, which is the standard W3C language for writing restrictions. The first method is implemented entirely within Neo4j, while the second method is implemented on an application level by using the Java programming language, where it is checked whether the user has entered a correct value according to the created constraint. In Neo4j, we decided to make one SHAPE which was named after the type of node, namely the *CourseShape*. In this case, the *CourseShape* represents the domain that the work requires.

The graph database created for this example will be shown below, and the SHACL syntax for creating the domain constraint will be explained, followed by a detailed presentation of the implementation in Neo4j and the Java programming language.

7 Validation

7.1 Creating a database for validation purposes

This section will describe how to create a database in the Neo4j system and how to implement a sample domain constraint. The idea is to create a domain that would limit the entry of ECTS points for a given course to the 1–8 value range. Any other value entered will result in an error. Prior to development, a data model was designed and developed for the sample exams database (Figure 4). The sample graph model includes exams, participants taking the exam and their exam grades, as well as other information required to schedule an exam deadline (lecture hall and professor).

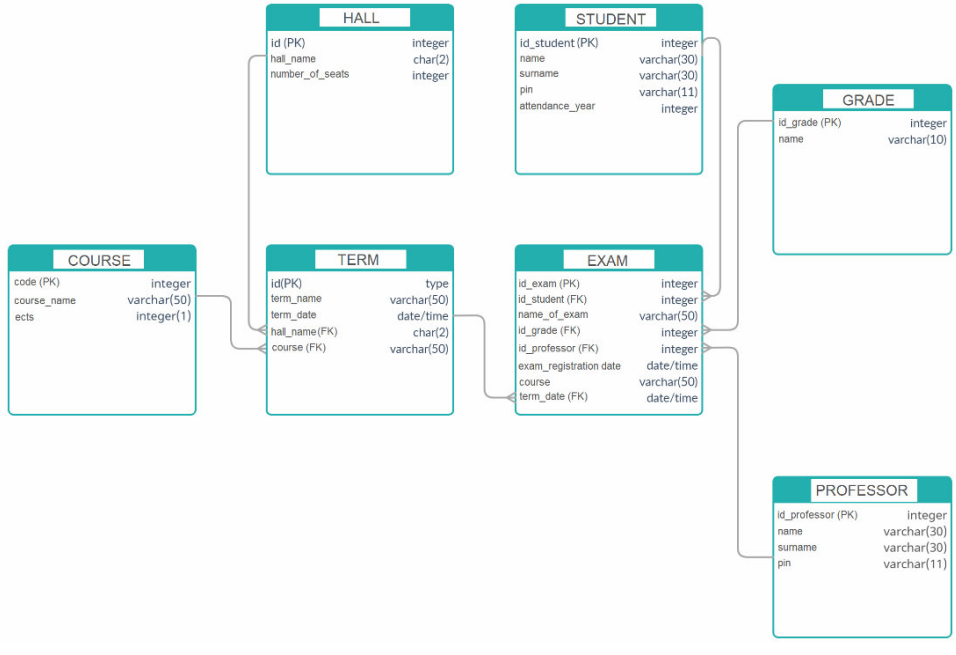
The data model can be represented as follows:

- Each student takes an exam from a course.
- This course is taught by a professor who is also the examiner.
- Each exam has a scheduled deadline and a hall where it takes place;

Each entity represents one type of node to be created, and each node has properties that describe it and by which edges are ultimately established. In total, there are 55 nodes in the database and 74 edges between these nodes. The ‘MATCH p = (n)-[e]-() RETURN p’

command gives the final state of the database, i.e., nodes and edges between nodes. Nodes and edges were created in the standard way, as explained in the following section. To make the edges as visible as possible, only a small part of the entire created database is presented, as can be seen in the following figure.

Figure 4 Data model for the database 'exams' (see online version for colours)



7.2 *Creating a domain constraint*

According to the Neo4j documentation (Neo4j, 2020), the first thing to do is to copy the jar file to the 'neo_home/plugins' DBMS directory. Once the jar file has been placed in the directory for the sample database called 'foi', one must go to the settings to change the configurations.

The 'neo_home/conf/neo4j.conf' configurations file must be changed with the following configuration added:

- dbms.unmanaged_extension_classes= n10s.endpoint=/rdf
- dbms.security.procedures.unrestricted=apoc.*
- apoc.trigger.enabled=true.

When the plugin is added, the server must be restarted to verify that the installation was successful. The list of procedures must contain those starting with n10s, which can be seen in Figure 8.

Verification is done via the following query:

```
call dbms.procedures ()
```

Figure 5 Overview of one part of the exam database (see online version for colours)

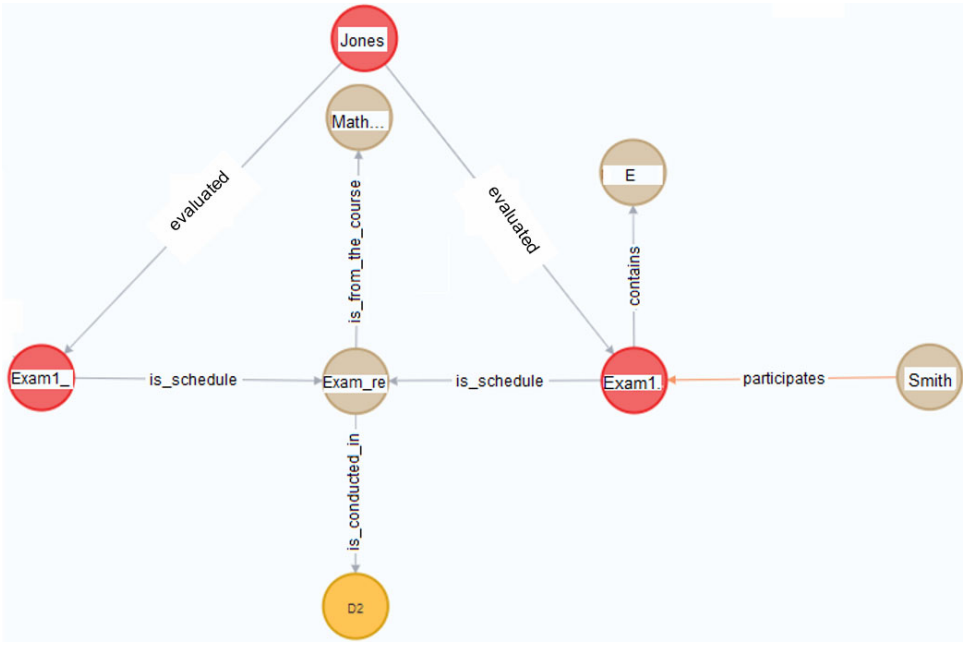


Figure 6 Inserting a jar file (see online version for colours)

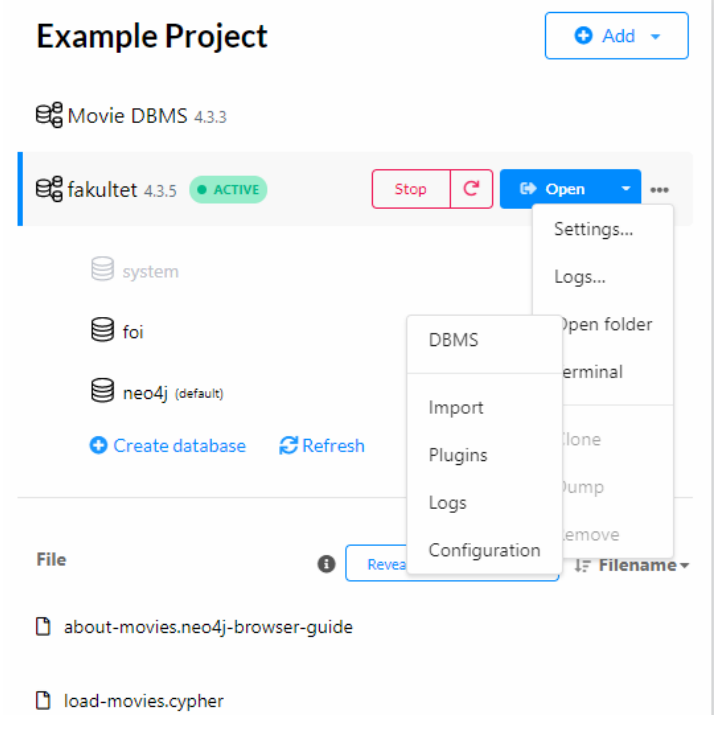


Figure 7 Configurations modification (see online version for colours)

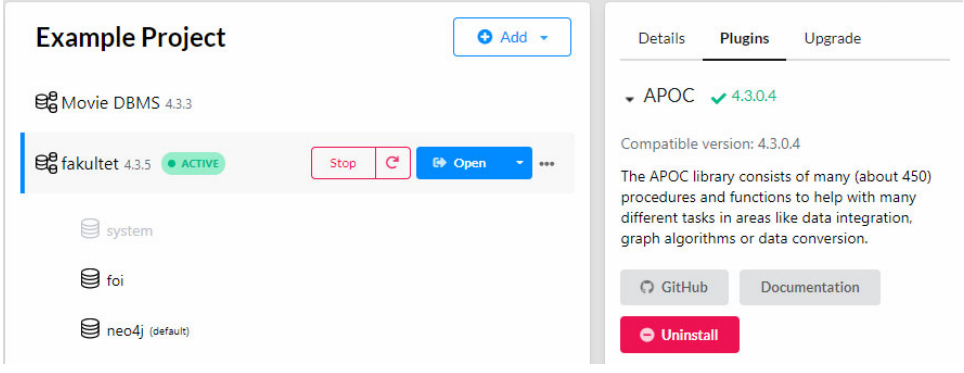
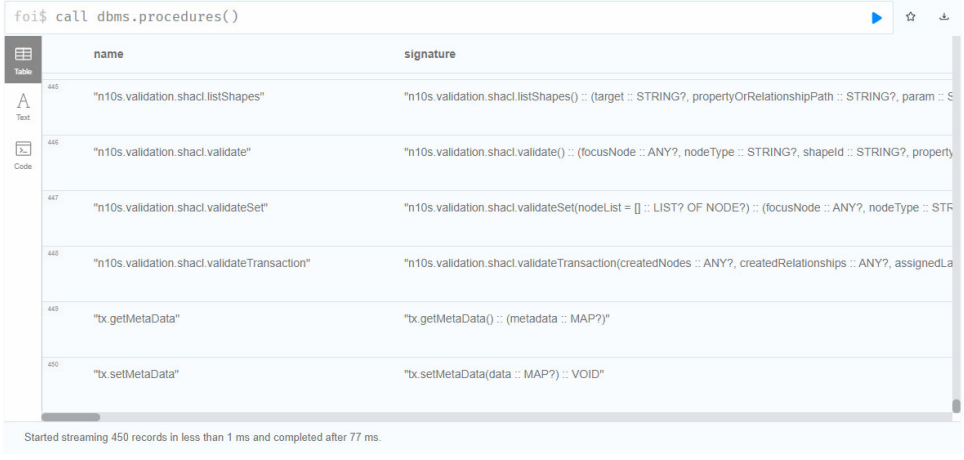


Figure 8 List of procedures registered for the sample database (see online version for colours)



After checking, it is necessary to initialise the configuration with the following query:

```
CALL n10s.graphconfig.init ({handleVocabUris: 'IGNORE'});
```

If there are nodes, this step is skipped, but if not, courses with their corresponding properties must be created. In the proposed case, it would be:

```
CREATE (Databases: Course {id: 1, title: 'Databases', ects: 5})
```

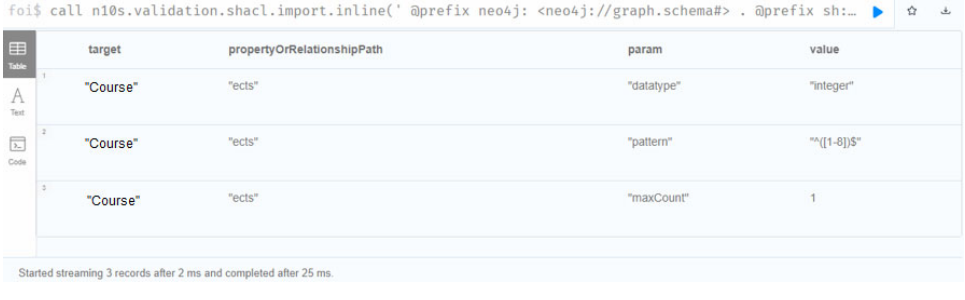
The next step to create a constraint is to create a schema. The database schema is tailored to the needs of each system. In our case, the schema will include a check on the value of the *ects* property entered by the user when creating a new course node. This custom schema was created directly in Neo4j, and it looks like this:

```
call n10s.validation.shacl.import.inline ('
@prefix neo4j: <neo4j: //graph.schema#>.
@prefix sh: <http://www.w3.org/ns/shacl#>.
```

```
neo4j: CourseShape a sh: NodeShape;
sh: targetClass neo4j: Course;
sh: property [
sh: path neo4j: ects;
sh: pattern '^ ([1-8]) $';
sh: maxCount 1;
sh: datatype xsd: integer;
];
.
',' Turtle ');
```

A SHACL constraint has been added to Neo4j. First, it was necessary to define the name of the shape that was created (e.g., shape *Course*). A constraint must then be placed on the node to which this constraint is applied. In this case, it is the *course* node. When a node is defined, it is necessary to specify the property to which the constraint is assigned, which is *ects* in this case. Once everything is defined, boundaries can be set.

Figure 9 SHACL constraint view (see online version for colours)



The condition is set that the entered value of ECTS must be between 1 and 8 (therefore, the user must not enter a value less than 1 or greater than 8). The next condition that must be met is the type of data that must be an integer (*sh: datatype xsd: integer*), and that only one number (*sh: maxCount: 1*) can be entered. The string pattern constraint was applied in our example to restrict the use of property *ects*. Value range constraints and property pair restrictions can also be employed if ECTS were an integer. Parameters and defined conditions can be seen in Figure 9.

Afterwards, the active shapes are checked using the command:

```
call n10s.validation.shacl.listShapes ()
```

The command will check and show all restrictions currently active in the database.

Once it is confirmed that the limits exist, it must be checked that all values are correctly entered. The entered values are checked as follows:

call n10s.validation.shacl.validate () yield focusNode, nodeType, propertyShape, offendingValue, resultPath, severity

If an ECTS value within the database is greater than 8 or less than 1, it will return those records to us. An example can be seen in Figure 10.

Figure 10 Checking for incorrect values (see online version for colours)

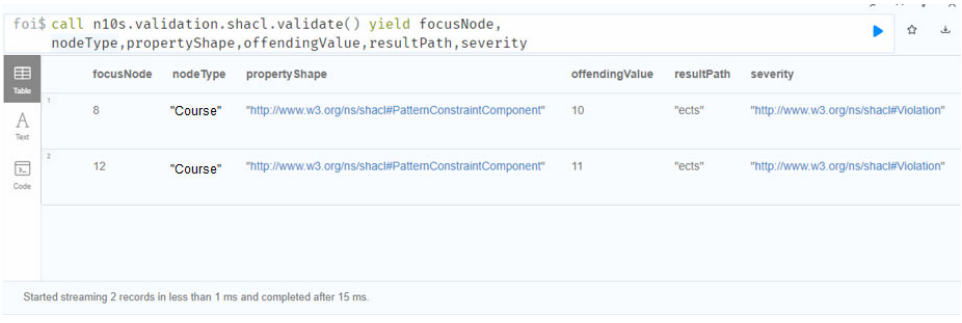


Figure 11 Display of correctly entered values (see online version for colours)

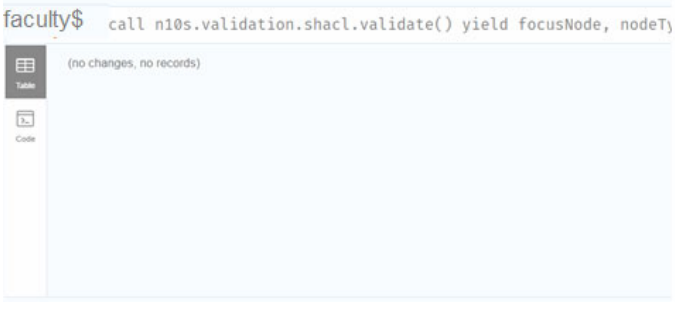


Figure 12 Creating a trigger (see online version for colours)

If all values are entered correctly, the following window will be displayed (Figure 11).

To prevent any errors and save all correct data to the database, one trigger was created that will be triggered before the transaction, regardless of the transaction being executed. The created trigger is defined in the following way:

CALL apoc.trigger.add ('shacl-validate', 'call n10s.validation.shacl.validate Transaction ($ createdNodes, $ createdRelationships, $ assignedLabels, $ removedLabels, $ assignedNodeProperties, $ removedNodeProperties, $ deletedRodelations, $ deletedRelationships, $ deletedNodes) (phase: 'before'})

Each trigger created can be paused, if necessary, which we can see in Figure 12.

Figure 13 Incorrectly entered ECTS property value (see online version for colours)

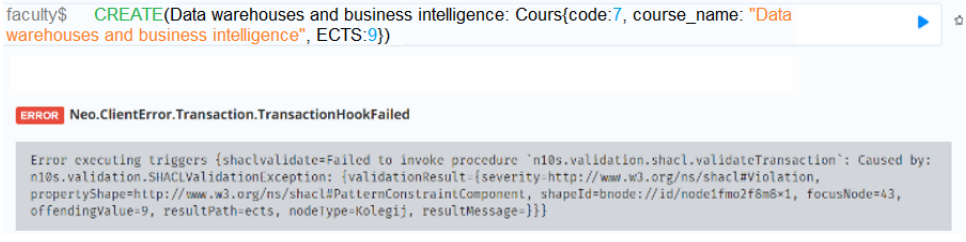
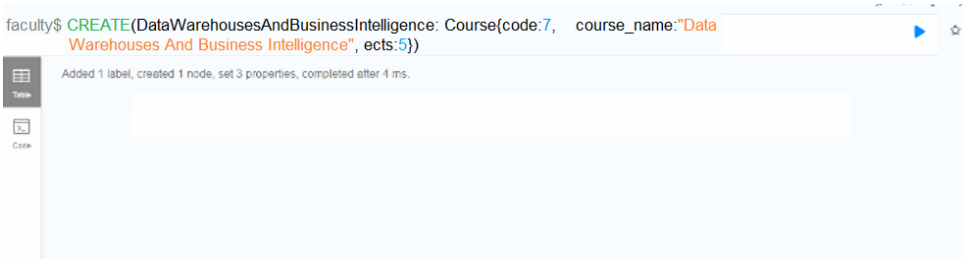


Figure 14 Correctly entered ECTS property value (see online version for colours)



Now, an attempt will be made to add a new course. The course will be called data warehouses and business intelligence and is the first example where the ECTS property will have a value of 9. Given that the entered value does not meet the limitations imposed by the domain constraint, Neo4j returns the following error:

The same course will now be entered, but value '5' will be added to the value of the ECTS, resulting in a successfully inserted node (Figure 14).

It can be seen that if an incorrect value is entered, it will immediately throw out the error and will not allow the incorrect value to be saved in the database. Still, if a value is entered that meets the permitted values; it will execute the command and save the data to the database.

Besides the database mechanisms, we can also use the Java programming language to enforce domains. In this case, the Neo4j driver and Maven were used for the implementation. Maven is actually an environment in which various projects are developed. It simplifies the construction process, as it is used to load libraries and add them to the application. The first step is to make a connection to the prepared sample database. Once the connection is established, an attempt is made to create a new node.

For this, it was necessary to create a method *createCourseCheckECTS*(), where two parameters were passed (course name and ECTS).

Figure 15 Creating a method in the Java programming language (see online version for colours)

```
public void createCourseCheckECTS(final String message,final Integer ects) {
    boolean ectsConstraint = false;

    if (ects >= 1 && ects <= 8){
        ectsConstraint = true;
    }

    if (ectsConstraint) {
        try (Session session = driver.session(SessionConfig.forDatabase("foi"))) {
            String greeting = session.writeTransaction(new TransactionWork<String>() {
                @Override
                public String execute(Transaction tx) {
                    Result result = tx.run( s: "CREATE (a: 'Course' {name: $message, ects: $ects}) " +
                                            "RETURN a.name + ', from node ' + id(a)",
                            parameters( ...keysAndValues: "message", message, "ects", ects));
                    return result.single().get(0).asString();
                }
            });
            System.out.println("You have successfully created a new node: " + greeting);
        }
    } else {
        System.out.println("Error, ECTS isnt in 1-8.");
    }

}
```

Figure 16 Saving courses to a database (see online version for colours)

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
pro 15, 2021 9:30:54 PM org.neo4j.driver.internal.logging.JULogger info
INFO: Direct driver instance 644460953 created for server address localhost:7687
You have successfully created a new node: Databases 1, from node 22
pro 15, 2021 9:30:59 PM org.neo4j.driver.internal.logging.JULogger info
INFO: Closing driver instance 644460953
pro 15, 2021 9:30:59 PM org.neo4j.driver.internal.logging.JULogger info
INFO: Closing connection pool towards localhost:7687

Process finished with exit code 0
```

Figure 17 Inability to save courses to the database (see online version for colours)

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
pro 15, 2021 9:31:40 PM org.neo4j.driver.internal.logging.JULogger info
INFO: Direct driver instance 644460953 created for server address localhost:7687
pro 15, 2021 9:31:40 PM org.neo4j.driver.internal.logging.JULogger info
INFO: Closing driver instance 644460953
Error, ECTS isnt in 1-8.
```

Before saving it to the database, it is checked whether ECTS are in the correct value range.

If ECTS is in the correct value range, it will return 'You have successfully created a new node: Databases 1, from node 22' as shown in Figure 16.

If an incorrect ECTS value is entered, i.e., a value is less than 1 or greater than 8, the error 'Error, is not in 1–8' will be returned, which can be seen in Figure 17.

8 Discussion

As a result of our research, it is clear that developing and integrating restrictions that may not yet exist in graph databases but are already supported in SQL databases is a viable area for additional study. This creates interesting opportunities for growing the landscape of constraints and enhancing the capabilities of our SHACL-based strategy.

There is space for improvement by adding additional constraints that are suited to certain nodes in addition to the limitations we have previously mentioned. We could expand this approach even though we now enforce a constraint on the ECTS attribute within the course node. We might impose constraints on the PIN property for students and instructors, requiring that it contain precisely 11 characters. Similar constraints might be put in place to prohibit users from providing numbers lower than 1 or more than 5 when specifying a year of study. This adaptability shows how our method may be able to handle various constraints conditions.

One exciting prospect for future use is the provision of domain limits through a user-friendly graphical interface. While this goal still informs our work as it develops, our present priorities are the conceptualisation, development, and validation of our method for building domains in the Neo4j graph database.

We highlight the dynamic nature of our SHACL implementation and demonstrate its flexibility to a wide range of real-world circumstances by recognising the capacity to add a variety of limitations beyond our initial scope. Because of its versatility, data validation has the potential to become more thorough and resilient, improving the precision and quality of graph data representation.

9 Conclusions

Graph databases are a very effective instrument/method for accessing and managing interconnected data. They become great solutions in many contemporary systems (like social networks) because of their capability to graphically represent complicated relationships between data and the potential to execute queries over big data sets at high speed. The ability to swiftly run queries and see data relationships is one of the critical advantages of graph databases over relational databases. Graph traversal allows for the effective querying of nodes and edges, allowing for the execution of complex queries, which may involve several joins in relational databases. A flexible schema is undoubtedly another crucial component of graph databases. One of the key distinctions between them and relational databases is their flexibility. Namely, users can continuously improve and modify their database models to suit their needs without wasting resources on costly schema migrations.

Graph databases do, however, have some limitations. The limited amount of constraints that can be employed is unquestionably one of the main drawbacks of these databases.

The primary focus of this study is on graph database constraints that still need to be met. It was suggested to use the so-called domains to impose new constraints. To avoid data redundancy, numerous constraints are specifically added at once while building a domain. In our example, the newly established domain had three constraints (data types, ranges, and counts). Our research is innovative, and the suggested implementation is practical and straightforward to apply if we follow the procedures outlined in the earlier chapters. A method involving the usage of APOC triggers, Neosemantics (n10s) plugin and SHACL verification were utilised to construct a domain in Neo4j. Graphs can be validated using SHACL following predetermined criteria that are met, and after that, they can be executed as part of the transaction commit. We gained the native ability to specify various constraints with the domain creation directly within Neo4j. APOC triggers were utilised to increase the implementation's safety and lower the likelihood of mistakes when communicating with the database. More specifically, we obtained an automatic check of the domain-defined limitations by using triggers that are triggered in the phase before the transaction's actual commit. The user needs to enter data into the database after the domain, SHACL validation, and APOC trigger have been correctly defined. All checks will be carried out automatically before the transaction is committed. In addition to the native implementation, it is demonstrated how the Java programming language can be used to implement the constraints outlined in the paper. Although constraints in graph databases are a very effective and significant element for maintaining data consistency, their implementation is very challenging. The methods for constructing constraints given in this study can serve as a foundation for future graph database research, which may even propose a specific syntax for creating domains in Neo4j that can be used in later system iterations. In the end, every additional constraint that is investigated will result in a rise in the stability and dependability of graph databases.

Acknowledgements

This work was funded by the Slovenian Research Agency (Research Core Funding No. P2-0057).

References

Carić, T. and Buntić, M. (2015) *Uvod u relacijske baze podataka*.

Chen, J., Song, Q., Zhao, C. and Li, Z. (2020) 'Graph database and relational database performance comparison on a transportation network', in *Communications in Computer and Information Science*, (Vol. 1244 CCIS, pp.407–418), Springer, https://doi.org/10.1007/978-981-15-6634-9_37.

Domdouzis, K., Lake, P. and Crowther, P. (2021) *Concise Guide to Databases: A Practical Introduction*, Springer Nature.

Fošner, M. and Kramberger, T. (2009) 'Teorija grafova i logistika', *Math.e*, Vol. 14, No. 1 [online] https://hrcak.srce.hr/41959 (accessed 6 November 2022).

Maleković, M., Rabuzin, K. and Šestak, M. (2016) 'Graph databases – are they really so new', *International Journal of Advances in Science Engineering and Technology*, Vol. 4, No. 4, pp.8–12 [online] https://urn.nsk.hr/urn:nbn:hr:211:997990 (accessed 6 November 2022).

Miller, J.J. (2013) 'Graph database applications and concepts with Neo4j', *SAIS 2013 Proceedings*, p.24 [online] https://aisel.aisnet.org/sais2013/24(accessed 6 November 2022).

Neo4j (2020) *Validating Neo4j Graphs against SHACL* [online] https://neo4j.com/labs/neosemantics/4.0/validation/ (accessed 5 November 2022).

Neo4j (2021a) *Constraints* [online] https://neo4j.com/docs/cypher-manual/current/constraints/ (accessed 5 November 2022).

Neo4j (2021b) *Syntax* [online] https://neo4j.com/docs/cypher-manual/current/syntax/ (accessed 7 November 2022).

Olivera, L. (2019) *Everything you Need to Know about NoSQL* [online] https://dev.to/lmolivera/everything-you-need-to-know-about-nosql-databases-3o3h (accessed 6 November 2022).

PostgreSQL (2021) *Create Domain* [online] https://www.postgresql.org/docs/9.5/sql-createdomain.html (accessed 6 November 2022).

Rabuzin, K. and Šestak, M. (2019) 'Creating triggers with trigger-by-example in graph databases', in *Proceedings of the 8th International Conference on Data Science, Technology and Applications – DATA*, ISBN: 978-989-758-377-3; ISSN: 2184-285X, pp.137–144, DOI: 10.5220/0007829601370144.

Rabuzin, K., Konecki, M. and Šestak, M. (2016a) 'Implementing CHECK integrity constraint in graph databases', in Suresh, P. (Ed.), *Proceedings of the 82nd IIER International Conference*, Berlin.

Rabuzin, K., Šestak, M. and Konecki, M. (2016c) 'Implementing UNIQUE integrity constraint in graph databases', in Westphall, C., Nygard, K. and Ravve, E. (Eds.), *Proceedings of The Eleventh International Multi-Conference on Computing in the Global Information Technology*.

Rdf4j (2021) *Validation With SHACL* [online] https://rdf4j.org/documentation/programming/shacl/ (accessed 6 November 2022).

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

Rohit, K. (2015) *Graph Databases: A Survey, Computer Science & Engineering*, Shiv Nadar University Greater Noida, India.

Sambolek, S. (2015) *NoSQL: pregledni rad*, Srednja škola Tina Ujevića Kutina, Kutina.

Stojanović, A. (2016) *Osvrt na NoSQL baze podataka – četiri osnovne tehnologije,* Polytechnic&Design, Tehničko veleučilište Zagreb, članak, Vol. 4, No. 1, Zagreb.

W3C (2017) *Shapes Constraint Language (SHACL)* [online] from https://www.w3.org/TR/shacl/ (accessed 25 May 2022).