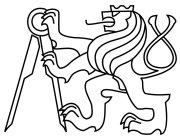
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Schema enforcement in a schema-free graph database I
Student:	Bc. Jiří Kovačič
Supervisor:	Ing. Michal Valenta, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2016/17

Instructions

The aim of the work is to explore possibilities of schema enforcement, i.e., specification and check of integrity constraints (IC) in a schema-free graph database (GD). A prototype is required in order to proof the concept. Selection of supported ICs will be based on analysis of practical requirements and will take into account the cost (in the sense of computational complexity) of ICs checking.
There are two parallel theses focused on this topic. They share the first part - requirements specification (point 1 below).
1. Explore and collect realistic requirements for IC enforcement in a GD.
2. Study GD neo4j and query language CYPHER, its implementation and possibilities of its extension.
3. Select a subset of requirements collected in point 1 that are convenient for implementation in CYPHER.
4. Design the syntax for definition of ICs selected in point 3 and the way of their implementation.
5. Implement functional prototype, realize testing, and evaluate your solution.

References

Will be provided by the supervisor.

L.S.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague November 18, 2015

Czech Technical University in Prague
Faculty of Information Technology
Department of software engineering

Master's thesis

Schema enforcement in a schema-free graph database I

Bc. Jiří Kovačič

Supervisor: Ing. Michal Valenta, Ph.D.

4th May 2016

Acknowledgements

I would like to express my gratitude to my supervisor of my Master's thesis Ing. Michal Valenta, Ph.D. for his time and valuable guidance, who was always available when I needed, and was able to share his rich knowledge with me. Furthermore I would like to thank MSc. Michal Bachman, the creator of GraphAware Framework, who has willingly shared his experience. I would like to thank Ing. Jaroslav Ramba for providing me a contact with MSc. Michal Bachman. I would like to thank Bc. Lenka Chmelíková, for being the grammar and spelling police, taking her time to carefully read the thesis. I would like to thank my parents for their patience and support. Finally, special thanks go to everyone at CTU FIT for providing me with scientific background.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 4th May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kovačič, Jiří. *Schema enforcement in a schema-free graph database I*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

V této práci jsou představeny možné druhy integritních omezení, které mohou být implementovány do grafové databáze Neo4j. Práce je rozdělena ve větší míře na teoretickou a v menší míře na praktickou část. Cílem teoretické části je vyvodit různá omezení týkající se integritních omezení a podložit tato odůvodnění samotnou použitelností i mimo jiné ve formě časové složitosti. Dále se v práci zabýváme návrhem nové syntaxe jazyka Cypher pro definování nových integritních omezení. V implementační části je navržen prototyp rozhraní, které má na starosti spouštět kontrolu dat v grafové databázi Neo4j na základě nadefinovaných integritních omezení.

Klíčová slova Neo4j, grafová databáze, integritní omezení, Cypher, schéma, validace, konfigurace schématu, GraphAware framework

Abstract

This thesis introduces all possible types of integrity constraints which can be implemented into Neo4j graph database. This work is divided into two parts, a theoretical and a practical. The aim of the theoretical part is to draw various restrictions relating to the integrity constraints and substantive reasons by the usability itself, inter alia, form of time complexity. Theoretical part also discusses and proposes a new syntax of Cypher Query Language for defining new integrity constraints. A prototype interface implementation is designed in the practical part with ability to start a validation process on the bases of defined integrity constraints on data in Neo4j graph database.

Keywords Neo4j, graph database, integrity constraints, Cypher, validation, schema, schema configuration, GraphAware framework

Contents

List of Figures

List of Tables

Introduction

A database management system (shortly DBMS) is a computer program (or collection of programs) that serves for storing, modifications and extracting information from a database, in general the DBMS takes care about management of information. Before the advent of databases were data traditionally organized in file formats [1]. The DBMS was a new concept that really helped with organization of data. The most widespread and well-known DBMS are relational databases. Their history start in early 1970s where the Relational Model was invented at IBM Research laboratories [2]. Relational databases using the model have become the most important systems for data management [3]. The model also called a *database Schema* describes a structure where data are organized into tables. Each table contains records where every single record with relevant data information is stored in a row, and each row has its unique identifier used for distinguish each other. Nowadays, the relational databases still belong between the most used DBMS in the world. However, currently DBMS are having trouble because of fast data growth. There are difficulties in semi-structured data which does not respect a rigid described structure and also due to data that can be sparse. All these things are not so good for relational databases and their limited performance of processing this kind of data. This fact that relational databases cannot handle new types of data opened the way for new types of database management systems. These new databases introduce themselves as NoSQL databases which provide a new mechanism of data processing. They do not use the relational model which is schema-specific but a specialize structure determined for such type of data the database should be able to manage. These databases can be *key-value*, *document* or *graph*. This thesis is engaged in graph databases.

Graph databases are young[1] NoSQL representatives that for data storing use a graph structure. The first thought of graphs and Graph theory falls within the early 18th century where Leonhard Euler introduced in his solution

[1] First commercial version was introduced in 2003.

of a problem the Seven Bridges of Königsberg. Graph databases do not use schema-specific model and they act as *schema-free* databases. The concept of *schema-free* means that it is not needed to define how the data should look like. We do not have to specify for example datatypes or how the data should be stored, and this is the main advantage of how to deal with sparse data. Graph databases use for data representation nodes and relationships. Connections between nodes are done by relationships. Both nodes and relationships can contain properties which is a key-value structure where under the specific key is stored a particular value. Nodes and relationships are separated by its use and kind of *ascii-art* description. Nodes are represented in *(parentheses)* and we can give a zero, one or more node names called *Label(s)*. Relationships are represented in *[brackets]* and must contain the naming convention called a relationship *Type*.

The actual real world situation requires that everything must be done, processed, executed or controlled immediately. This enormous pressure from all sides is unsustainable and technology itself is driven into the ground-breaking heights. Graph databases are starting to be widely used for recommendation systems or for detection of fraud. These requirements demand almost real-time interaction where the getting of an important information does not take hours or minutes, but seconds or better milliseconds. Thus, one of the most important requirements is to keep low-time latency.

This diploma thesis focuses on some kind of database restriction and a validation where a control of these restrictions takes more time than is adequate and widely depends on the size of a database. Graph databases are normally schema-free which means that there are no restrictions. However, customers working with graph databases would like to use some techniques that are available in the relational databases that can restrict a database schema. This kind of restriction is called *Integrity constraint*. Integrity constraints define the database schema which can be afterwards enforced at database data.

Chapter 1 of this thesis introduces background of a research including Graph theory, what is a graph traversal and what types of NoSQL databases exist. In this chapter it is introduced graph database Neo4j which is used for design and implementing of those integrity constraints that are be convenient to deal with them. This chapter also includes basics of Cypher Query Language and its syntax which is in this work extended about a new integrity constraint definition and also there is mentioned a *GraphAware framework*, a tool that helps with advance Neo4j use cases. Chapter 2 deals with a research and how the integrity constraints are implemented in the relational databases. This chapter takes an inspiration that can be helpful for definition of integrity constraints in Neo4j. Chapter 2 also describes a problem of the integrity constraints. There is introduced a working paper called *Request for proposals: Cypher Schema syntax* designed and proposed by Cypher Language Group. The working document contains all possible integrity constraints that should be implemented with a provided description of their behavior. The last sec-

tion of the chapter 2 is a complete revision of integrity constraints. There is discussed possible problems and difficulties that may arise. Revision section deals with time complexity with combination of DML operations; it contains one of the possible suggestions of the new Cypher syntax for the integrity constraint definition and a discussion around the problem. Chapter 3 introduces a new Cypher Query Language syntax design for integrity constraints. The syntax itself is designed gently to keep a basic idea of actual Cypher syntax for querying Neo4j graph database. Designed syntax covers all possible options from creation of integrity constraint, its detection by Cypher MATCH clause, modifications by SET clause and other provided actions. Chapter 4 presents a prototype implementation called *SchemaConfigurationAPI* and how the integrity constraints should work and be handled in Neo4j graph database. Chapter 5 contains benchmark of implemented integrity constraints that are tested on a Cineasts movie database. Provided benchmark could not be compared with actual Cypher syntax in Neo4j because there are actually not supported such processes that are tested. However, the benchmark provides valuable information of implemented integrity constraints and how to use this information to the future. Chapter 6 comments on the evaluation and contains next proposal topics for further research in this area of implementing integrity constraint for Neo4j graph database.

Chapter 1

State-of-the-art

1.1 Graph theory

History of Graph theory goes back to the early 18th century where Leonhard Euler introduced in his paper a solution of a problem the Seven Bridges of Königsberg [4, 5]. The city of Königsberg in Euler time was in Prussia, today is in Russia and the city actual name is Kaliningrad. The city was set on both sides of the Pregel River and included two large islands which were connected each other by seven bridges [6]. Leonhard Euler solved the problem whether it was possible to find a path that every bridge could be crossed once and the starting and ending point of the walk must had been at the same place. In his work Leonhard Euler proved that this problem does not have a solution and the Graph theory had been born. Nowadays, we can see graphs everywhere. The graphs consist of points which are connected each other by lines. These points we can imagine, for example, as cities and a connection between them we can consider roads. But, this real-world example has its theoretical thought in nature science, Mathematics.

In this section we will introduce basic definitions from Graph theory which is a good basis for understanding how the graph databases work. All these definitions within the Graph theory section are taken from Diestel [7], unless indicated otherwise.

Graph A graph is a pair $G = (V, E)$ of sets satisfying $E \subseteq (V \times V)$. The elements of V are the *vertices* or *nodes* of the graph G, the elements of E are its *edges* or *relationships*. For each edge $e \in E(G)$ applies that it is connected with a pair of nodes u, v, where $u, v \in V(G)$. In a Graph theory literature we meet with terms *vertices* and *edges*, but instead in a graph database terminology we use *nodes* and *relationships*. These terms in this thesis are interchangeably used.

Order The *order* is the number of vertices of a graph G. Its number of edges is denoted by $\|G\|$. Graphs can be *finite* or *infinite* due to their order. In this text we consider all graphs as *finite*.

Incident vertex A vertex v is incident with an edge e if $v \in e$; then e is an edge at v.

Vertex degree The set of edges vertex v is incident with is denoted with $E(v)$. The degree of a vertex v, denoted as $d(v)$, is the number $|E(v)|$ of edges incident with the vertex v.

Subgraph Lets have two graphs $G(V, E)$ and $G'(V', E')$. The graphs G' is a subgraph of G, written as $G' \subseteq G$, if $V' \subseteq V$ and $E' \subseteq E$. That means that all vertices used in the graph G' are vertices of the graph G and all edges in the graph G' are the edges of the graph G. This definition was adopted from Bachman and Troup [8, 3].

Path A path is a *non-empty* graph $P = (V, E)$ of the form where $V = v_0, v_1, \ldots, v_n$ and $E = (v_0v_1), (v_1v_2), \ldots, (v_{k-1}v_k)$ and k is a finite number. The pair of vertices $(v_{k-1}v_k)$ is an edge and a number of edges $|E|$ in the path is its length.

Directed graph A directed graph or digraph $DG = (V, E, init, ter)$ is a graph $G = (V, E)$ with the maps additional information *init* and *ter*. Map init: $E \to V$ assigns every edge e an initial vertex $init(e)$ also called as a start vertex or node. Map ter: $E \to V$ assigns every edge e a terminal vertex $ter(e)$ also called as an end vertex or node. The edge e is said to be directed from $init(e)$ to $ter(e)$. Note that there may be several edges between the same two vertices. Such edges with the same direction are called *multiple* or *parallel edges*. If $init(e) = ter(e)$ the edge is called *loop*. This definition was adopted from Bachman and Troup [8, 3].

Vertex degree revisited In direct graphs the degree of a vertex v, denoted as $d(v)$, can be formulated as indegree and outdegree. Indegree, denoted $d_{in}(v)$, is defined as the number $|E_{in}(v)|$ of edges incident with the vertex v, where $E_{in}(v) \subseteq E(v)$. Similarly, outdegree, denoted $d_{out}(v)$, is defined as the number $|E_{out}(v)|$ of edges incident with the vertex v, where $E_{out}(v) \subseteq E(v)$. Note that a loop case adds one to both, the indegree and outdegree of its vertex v. This definition was adopted from Bachman [8].

Graph traversal

Graph traversal or graph search is in computer science a technique of visiting each vertex in a graph. Traversing a graph means visiting its vertices or nodes following the edges or relationships. There exist two most used algorithm to traverse the graphs, Depth-first search (DFS) and Breadth-first search (BFS). These algorithms search in linear time, respectively in $O(|V|+|E|)$, where $|V|$ is a total number of vertices and $|E|$ is a total number of edges. Both algorithms start searching from a root node. Figure 1.1 shows how traversal algorithms work. The Depth-first search starts from the root and goes through the nodes as quickly as possible to the leaf nodes. The Breadth-first search algorithm

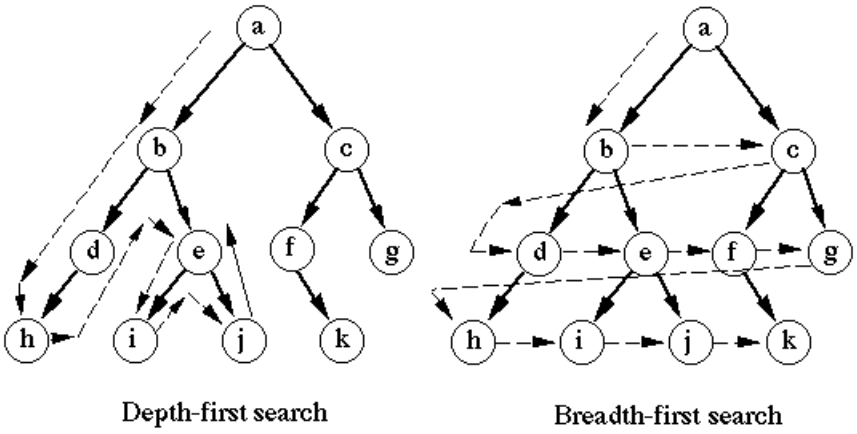


Figure 1.1: DFS and BFS traversing process [9].

starts at a top level (i.e. root node) and firstly explores its neighbor nodes before going to the lower level [9, 10].

1.2 Graph databases

Generally, databases are instruments which help us with storing and retrieving data. It is an organized collection of schemas, tables, queries, views and other database objects [11]. The whole functionality handles a Database Management System known as DBMS. The DBMS is a set of collaborative computer applications which interacts with a user, other applications and the database itself. The main objective of DBMS is to allow create, read, update and delete operations (or CRUD), querying and not least a database administration. The reason why graph databases began to come to foreground is that from year to year is increased the amount of data in the world and there is a need to effectively store and query them. These data are increasingly structured and also interconnected and we start to feel that relational databases have some limits and their capability may not be sufficient [12].

The databases can be divided into two categories, SQL and NoSQL. The SQL is an abbreviation meant for Structured Query Language. Development of the relational databases, thus databases using SQL started in 1970s at IBM laboratories [2]. The principle of the relational database is storing data into structured tables. Tables are composed of columns and each column has defined its datatype. The data itself are then stored into the rows where each row has such a unique identifier. Data across tables can be linked via keys. For querying data is used the SQL language mentioned above.

The graph databases belong between NoSQL databases. In comparison with the relational databases the graph database are quite a bit younger, with first implementation in 2000s [3]. What exactly describes NoSQL covered

Michael Hunger in his presentation [13]: "NoSQL describes ongoing trend where developers increasingly opt for non-relational databases to help solve their problems, in an effort to use the right tool for the right job." NoSQL databases can be arranged into specific subcategories reflecting their storage policy.

- **Document stores** are document-oriented databases offering storing, retrieving and managing document-oriented information also known as semi-structured data [14]. The behavior is similar to a filling cabinet in medical clinics where doctors have stored documents about patients in their medical folders. Typical representatives are MongoDB, CouchDB and OrientDB.

- **Key-Value stores** are databases based on a global key-value mapping. It is a table with a column of keys and a column of values, so all data are stored within this table [3]. Key-value stores act like large, distributed hashmap [15]. Between representatives belong Membase, Riak, Redis and Amazon DynamoDB [13].

- **Graph databases** store data in a graph. Data information is stored in both nodes and relationships. A graph database is an online, operational database management system with CRUD operations and is considered as a schema-free database [15]. Graph databases are those graph databases that require edge transition in constant time $O(1)$. Graph databases store data in denormalized form against relational databases where is needed to perform join operations, but this is paid by higher time complexity for writing operations [16]. Typical representatives are Neo4j, Sones, Infinite Graph, AllegroGraph and FlockDB [12].

Of course, we are not able to mention all representatives of NoSQL databases because a development trend to obtain new kinds of graph databases is still running.

Property graph

In section 1.1 we mentioned how the graph is defined. The graph is a pair $G = (V, E)$, where letter V is meant for *vertices* and letter E for *edges*, $E \subseteq (V \times V)$. However, this kind of graph is not convenient to store suitable data and also we are not supposed to want it because the graph does not have assumptions and cannot behave as a suitable data model for data storage [8]. The Neo4j solved data model for graphs itself and uses a *labeled property graph* model. The label property graph must fulfil following characteristics [15]:

- Property graph contains nodes and relationships.

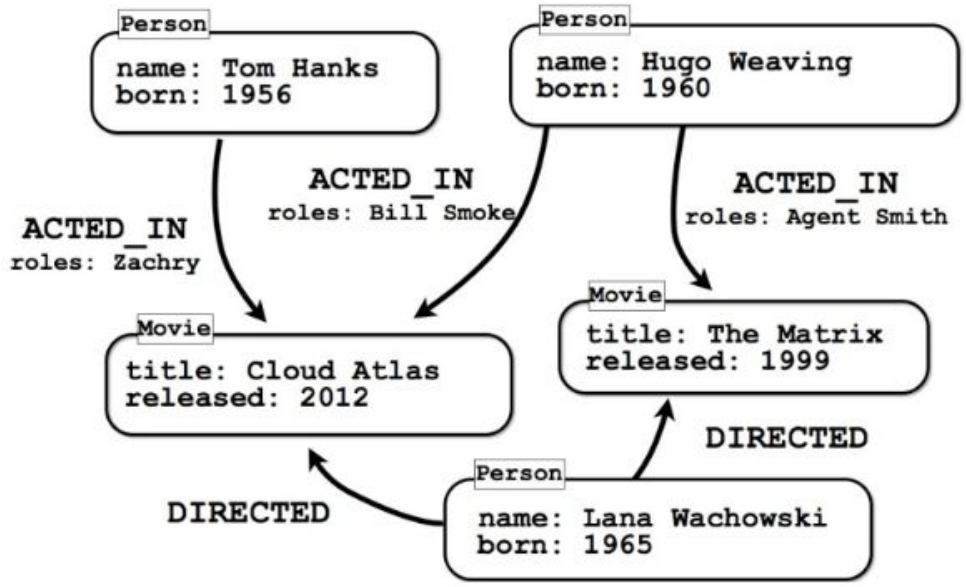


Figure 1.2: Labeled property graph movie example [17].

- Nodes can be labeled with one or more labels which is good for identification.
- Nodes can contain properties; properties are stored as a key-value pairs.
- Relationships can be named and directed, the name of a relationship is called a Type.
- Relationships have both a start and end node.
- And relationships can also contain properties like nodes.

Figure 1.2 is an existing example which serves as a basic example to explain how the graph should look like, or how should the information be kept in Neo4j graph database. The graph stores information about movies and persons where a particular relationship distinguishes a person from a director or an actor. Figure 1.2 fulfills all mentioned characteristic describing the labeled property graph.

Neo4j

Neo4j is a graph database from the NoSQL category. It is an open-source project developed in Java and Scala languages running on the Java Virtual Machine [3]. In contrast with the relational databases the Neo4j graph database is schema-free which means that there are no limitations to adhere any

predefined database structure, i.e. a database schema [8]. Neo4j is suitable for Highly connected data where belongs Social networks, or Recommendations, Business Intelligence, etc. Neo4j offers many features [18]:

- For all transactions supports full ACID behavior. ACID is an abbreviation for *atomicity* – a transaction must finish as the whole, if some part of the transaction fails the entire transaction fails and a database state will be unchanged. *Consistency* ensures that database is valid before and after the transaction. *Isolation* – parallel transactions will not affect each other and *durability* is for transactions that if they were once committed, then the information was permanently stored in the DBMS.
- Database is provided in two modes, an embedded and server mode. Embedded means that the database is tightly integrated with a certain application software. It is faster, but we are not able to share it; no other process can access the database. Whereas the server mode is a little bit slower, but the advantage is in its sharing [19].
- Database guarantees high speed querying. This is due to Neo4j design for storing data into disk. The idea itself is in a smart representation of nodes and relationships.
- High availability. This feature is only available in Neo4j Enterprise Edition. It allows two main features, a *Database as a fault-tolerant system* and *Database as a scalable system*. In the first option we can imagine a master database with many replicas which can cover all sorts of mistakes. The second option allows database to be clustered on multiple physical servers, this is known as a horizontal scaling [20].
- Cypher is supported. Cypher is a textual, declarative query language built to simplify querying under the graph database. Cypher uses a form of *ASCII art* to describe a pattern to query the database. Declarative paradigm means that we specify what we want to obtain, but not how to something obtain.

Architecture

Neo4j acts as a *Native Graph Storage*. We can imagine this phrase that a graph database has a feature called index-free adjacency. Thus, each node acts as a micro-index and maintains direct references to its adjacent nodes. These micro-index nodes replace global indexes which link nodes together, the global index is known for a nonnative graph databases [15]. With index-free adjacency one traversing step, step among two nodes, is constant, in time complexity $O(1)$.

Figure 1.3 shows the whole architecture concept for Neo4j graph database. It is split into three parts, JVM offering variety of APIs, OS with file system

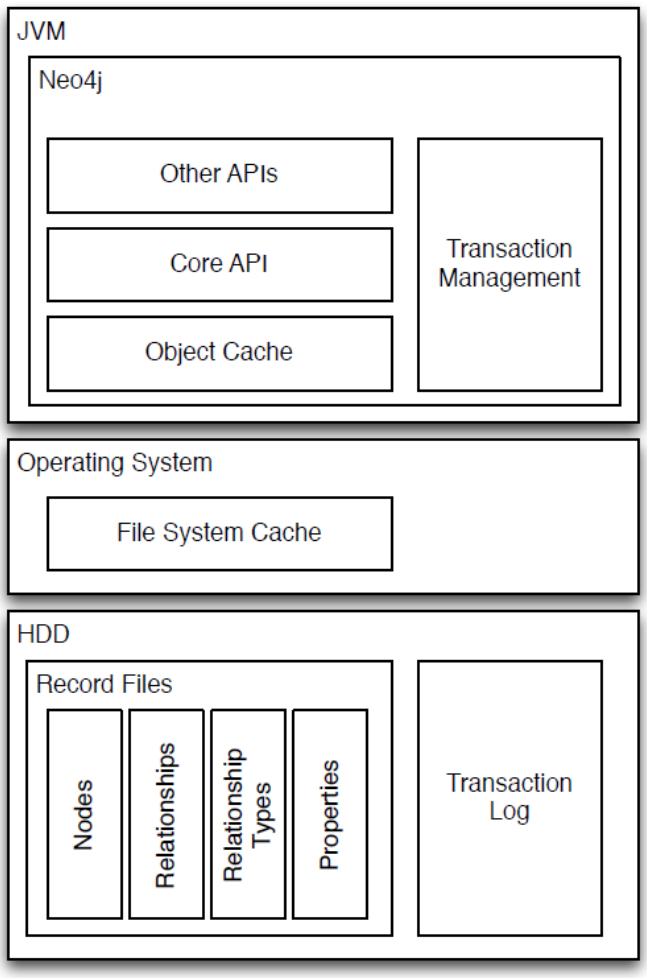


Figure 1.3: Neo4j High-Level Architecture [8].

cache and HDD where the particular functionality is designed. Due to index-free adjacency must be appropriately proposed a physical storage space. That means that each file record is stored separately on a disk. Figure 1.3 distinguishes special files for nodes, relationships, relationship types and properties for both nodes and relationships. Figure 1.4 shows two records, how nodes and relationships are physically stored respectively.

Node store file

The node store file is determined for storing node records. This applies for each node which is created. The node store file is fixed at its agreed size. Each node has reserved 15 bytes. This brings a very great asset where the fixed-size enables us a fast lookup for certain node in the node store file

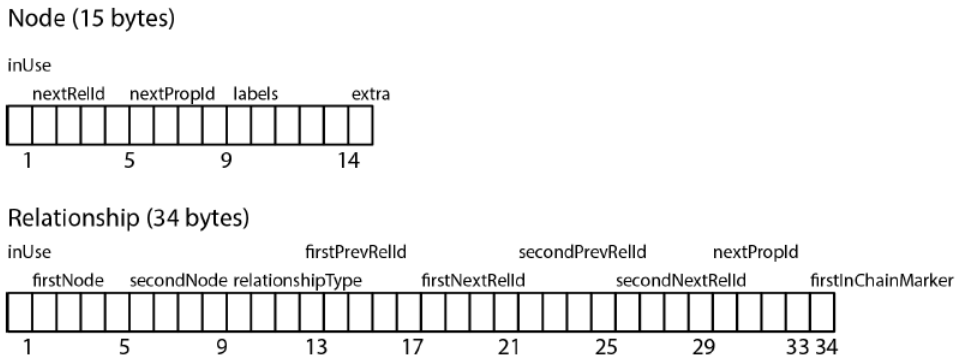


Figure 1.4: Neo4j file record structure for a node and relationship [15].

because we simply by a node identification number can calculate where the node physically lays in the file. If we have reserved 15 bytes per node and want to find a 50th node (its ID) we will simply find the node location with a calculation $15 * ID$, this leads to the $O(1)$ complexity time. The first byte is an in-Use flag which tells us whether the node is still active or if the database can re-write it with a new one. The next four bytes are reserved for first relationship ID connected to the node. Another next four bytes represent ID for the first node property. The five bytes are served for labels, i.e. how the nodes are named, and the last byte is reserved for flags [15].

Relationship store file

The relationship store file stores relationship records. As well as the node store file, the relationship store file is also fixed-size. Each relationship record has reserved 34 bytes length. The first byte is in-Use flag with the same behavior like in the node representation. The next pair of four bytes is for IDs of the nodes, the start node and end node of the relationship. The next four parts consist of 4 bytes are reserved as pointers for the next and previous relationships records for each of the start and end node. The last byte is a flag called relationship chain which is used in Neo4j's traversal framework [3, 15].

Cypher

Cypher is a declarative pattern matching query language allowing querying and updating of the graph. The declarative expressing means that we describe thorugh Cypher what we want to get from the graph instead of how we want to get from the graph which is a sign for an imperative paradigm [21]. The Cypher idea itself was to become a human-readable. It is illustrated in a pattern definition where is used a form of *ASCII art* to represent the graph pattern in the graph database. Cypher borrowed many statements from other query languages like SQL and SPARQL. Cypher uses clauses such as [22]:

- CREATE for a node and relationship creation.
- MATCH for a problem description.
- RETURN for a returning the result set.
- WHERE, ORDER BY, WITH, MERGE, SET, REMOVE, INDEX, CONSTRAINT, etc. whose name says enough.

Among another query languages which supports graph database Neo4j belong RDF query language *SPARQL* and the imperative, path-based language *Gremlin* [15]. This work deals with Cypher. Our goal is to extend the Cypher with new functionality supporting our constraint definition entries which is discussed in a chapter 3 **Cypher syntax design**.

Constraints

Integrity constraints are rules which enforce data structure in a database schema. These rules specify various restrictions that the database must fulfill, if something violates the constraint rule, then the database changes are not allowed. Integrity constraints are widely used in the relational databases. The integrity constraints have developed into the present form over the relational databases lifetime. The more about integrity constraints is discussed in section 2.1 **PostgreSQL and Oracle Constraint Analysis**.

Neo4j graph database supports very small amount of integrity constraints. Neo4j is divided into two versions, a Community edition and Enterprise edition which is affordable for a fee. Community edition supports Unique constraint which means that it is possible for a certain labeled node with a particular property assert its unique value, but it is not supported covering multiple properties simultaneously yet, and also uniqueness is not supported for the relationship properties too. During writing this diploma theses was announced a new version of Neo4j (exactly v2.3) where in Enterprise edition was added a new integrity constraint feature called *Property Existence Constraints* [23]. This integrity constraint lets us to specify a rule which will watch if the property is satisfied, i.e. must be set in a given node or relationship. The property existence constraint is an analogy to *NOT NULL* in the relational databases. Examples below (1.1) show us a current state of integrity constraint definition in Neo4j graph database [23]:

Listing 1.1: Integrity constraint definition in Neo4j GD

```
(1) CREATE CONSTRAINT ON (movie:Movie) ASSERT
                                movie.title IS UNIQUE

(2) CREATE CONSTRAINT ON ()-[like:LIKED]-() ASSERT
                                        exists(like.day)
```

First integrity constraint rule checks property uniqueness for all nodes with a label *Movie*, this kind of integrity constraint is possible to create in both editions the Community and Enterprise. The second proposal is an example how to validate a property existence where the use is shown on the relationships. All relationships with a type *LIKED* must contain a property day. This integrity constraint feature is only available in the Enterprise edition.

The next tasks of this diploma thesis is to design and implement extension for Cypher language which covers some of the integrity constraints specified in chapter 2 **Analysis** section 2.1 and revised in section 2.3, respectively.

GraphAware framework

GraphAware Framework is an open-source convenient platform for extending Neo4j graph database. This framework was introduced in Bachman's MSc thesis *GraphAware: Towards Online Analytical Processing in Graph Databases*. The main target of GraphAware Framework is to speed-up development with Neo4j by providing a platform for building useful generic as well as domain-specific functionality, analytical capabilities, graph algorithms, and many more [24]. GraphAware Framework has two key parts of functionality [25]:

- **GraphAware Server** is a Neo4j server extension module allowing to rapidly build (REST) **API**s[2] on top of Neo4j using Spring MVC[3].
- **GraphAware Runtime** is a runtime environment for both embedded and server deployments. GraphAware Runtime helps us with the use of pre-built as well as custom modules called *GraphAware Runtime Modules*. These modules can extend a core functionality of Neo4j graph database in ways such as:
  - Enforcing the specific integrity constraints on the graph schema.
  - Maintaining an in-graph index.
  - Improving performance by building in-graph indices and much more.

Neo4j graph database has implemented techniques which help developers to create their own plugin extensions, but these techniques in certain parts are not sufficient. There is a solution in GraphAware Framework which was designed to help developers with advanced usage of Neo4j graph database. The framework deals with the following use cases [3]:

[2]REST APIs are such APIs whose methods can be accessed via HTTP methods [3].

[3]Spring MVC is a Java framework that, among other things, implements model-view-controller pattern [3].

- **Custom APIs** is in Neo4j known as *Unmanaged Extensions*. This helps us to build new functionality with full control over API. It is one level below than creating *Server Plugins* where the API is not needed [26]. With help of GraphAware Framework we can simply develop such APIs for miscellaneous objectives including functions that are missing in Cypher query language, database access restrictions or custom-build input/output formats [3].

- **Transaction-Driven Behavior**, Neo4j provides behavior to hook into a transaction handling process and inspect these transactions [3]. This means that Neo4j gives us a chance to react to these transactions right before they are actually committed, or right after they have been committed [27]. Such behavior can be useful in cases like notifications of modified data, additional modifications or schema enforcement. GraphAware Framework provides suitable API to work easily with transactions [3].

- **Asynchronous Computation** is a very useful thing when we want to run operations in the background of Neo4j. GraphAware Framework offers to build functionality that can be executed in the background. Asynchronous Computation can be widely used in precomputing recommendations, gathering information or statistics [3].

Since the time GraphAware Framework were developed came many modules with useful features. Such features for example are [25]:

- **GraphAware Test** is a module which provides a simple code testing with interaction with Neo4j graph database. The code can be tested in three possible ways:
  - **GraphUnit** is designed for easy unit-testing of code.
  - **Integration Testing** is a part which is helpful for integration testing of Neo4j-related code.
  - **Performance Testing** which is good to measure performance of Neo4j-related code, how much time consuming our operations are, respectively.

- **Improved Transaction Event API** where the main idea was introduced in the paragraph **Transaction-Driven Behavior**.

There are much more examples of implemented modules which are available, described, documented and ready to use in GraphAware Framework which has free access in a GraphAware Git repository [25].

Chapter 2

Analysis

2.1 PostgreSQL and Oracle Constraint Research

This section discusses the possibilities of integrity constraints in Oracle and PostgreSQL relational databases. According to this chapter there are lifted up substantial ideas which were implemented by Oracle and PostgreSQL. The next chapter dealing with integrity constraints compare main ideas for Neo4j graph database with this part of the work.

Relational databases use tables for data storing. These tables as parts of a database schema needs to be created which serves Data Definition Language (DDL). Figure 2.1 shows a whole life-cycle creation process for defining a new table in a database. At first, we need to create a Table definition and Instance. Then we need to define columns for the particular table and after column definitions we must define the integrity constraints for the table. Dashed rectangles are optional and cover possibilities needed to a table update operation [28].

Integrity constraints are rules used to specify some limitations to the certain database, basically determined for create, insert, update and delete operations. The integrity constraints define a border to fulfil database correctness. Integrity constraints consider right data from the other ones. Suitable data are

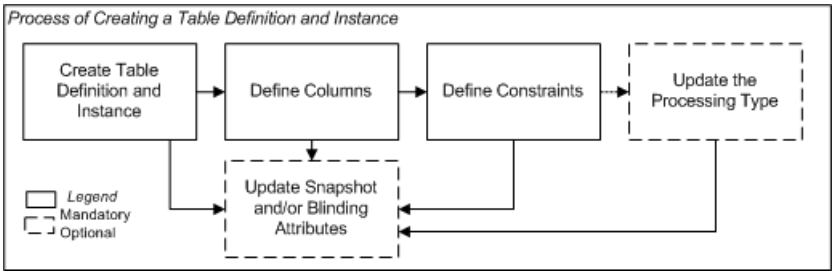


Figure 2.1: Oracle Process of Creating a Table definition and Instance [28].

stored and data which violate the integrity constraint condition not [29]. Why do we really need integrity constraints, or why are we trying to implement them? One possible answer is that we just simply need those integrity constraints because sometimes we need our data to look somehow as we wish. Thus integrity constraint tells us how the data should look like in storing them into the tables or columns [30]. Generally, the integrity constraints can be divided into three sections [29, 31]:

1. **Declaratively** (Server-side)

   Integrity constraints defined as a declarative on the server-side tell us that they are stored together with a database schema into a database machine. This way has both advantages and disadvantages. The advantage is that integrity constraints are defined on the database server and during the execution DML, or DDL operations this database server validates input data. The one big difficulty is the server itself and its validation. This is being done because the user working with the database is informed with error notifications with some delay. Delayed communication with the database server can be pesky and can lead to inefficiencies during a work and a bad time scheduling, respectively.

2. **Procedurally** (Client-side)

   The next possibility is defining client-side procedural integrity constraints. From a user perspective who works with relational database all time is this way very effective. It is given by integrity constraints which are physically stored on the client machine. This causes immediate reaction for data input and no time delay. But this approach has one flaw. If we needed a database which must be running for many applications, then the integrity constraint control procedures would have to be defined for every single application. Also there is no guarantee an intervention to the database schema. Also the intervention to the database schema is not ruled out.

3. **Procedurally** (Server-side)

   The control procedures and integrity constraints are saved and run on the database server machine. This job does the triggers. The triggers are small programs which are executed depending on a particular event. These events are DML operations like DELETE, INSERT and UPDATE and DDL operations such like CREATE, ALTER and DROP statements.

All these three methods have its pros and cons. Among the best and most effective way to introduce integrity constraints in the relational database is to use the declarative definition, accompanied by one of the procedural definitions [31].

The relational databases fulfilling SQL92 standard where we can include also Oracle and PostgreSQL databases. In general, there are defined these integrity constraints [29, 32]:

- NOT NULL,
- UNIQUE KEY,
- PRIMARY KEY,
- FOREIGN KEY,
- CHECK.

Integrity constraints can be defined in both ways, either at a column level or at a table level. Of course, we have to admit that in a creation process at the table level where columns are defined are also entered datatypes for those columns. We automatically count on with this integrity constraint, but in comparison relational databases with Neo4j graph database where is no need to use datatypes is this informal constraint needed which must control and validate datatypes. For example if we define a column with an INTEGER datatype, it will not be possible to place there a STRING value.

Integrity constraints mentioned above have same behavior in both databases Oracle and PostgreSQL. There is only one small different part and it is their definition or a syntax statement. If the mentioned basic integrity constraints are not enough, then exist a possible way how to enrich the database with small programs called triggers.

NOT NULL integrity constraint

Integrity constraint called NOT NULL is always defined at the column level of the database. The NOT NULL integrity constraint ensures that there will not be stored any null values on the database. Listing 2.1 shows a usage of the NOT NULL integrity constraint defined in more than one column.

Listing 2.1: PostgreSQL NOT NULL constraint definition [32]

```
CREATE TABLE products (
        product_no integer NOT NULL,
        name text NOT NULL,
        price numeric
);
```

The NOT NULL integrity constraint is functionally equivalent with creating of check integrity constraint **CHECK(columnName IS NOT NULL)** [32], but in SQL databases are explicit not-null integrity constraints more efficient due to their performance.

UNIQUE integrity constraint

In Oracle database a UNIQUE integrity constraint is known as *Unique key* [28]. The UNIQUE integrity constraint ensures that inserted data in a column, column where is a definition with the Unique integrity constraint, in all rows in the table must contain unique column values. The UNIQUE integrity constraints can be written in both ways as a column and as a table statement which is shown in listings 2.2 and 2.3. The UNIQUE integrity constraints should also be defined in more columns than one in the table.

Listing 2.2: PostgreSQL UNIQUE constraint as a column definition [32]

```
CREATE TABLE products (
        product_no integer UNIQUE,
        name text,
        price numeric
);
```

Listing 2.3: PostgreSQL UNIQUE constraint as a table definition [32]

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric,
        UNIQUE (product_no)
);
```

PRIMARY KEY integrity constraint

A PRIMARY KEY integrity constraint is used as a unique identifier for rows in a table. By the PRIMARY KEY integrity constraints we can determine unambiguous set of information in the table. The PRIMARY KEY is usually set to one or more columns for proper data recognition.

The PRIMARY KEY integrity constraints also can be written in both ways as a column and table statement like a UNIQUE integrity constraint which is mentioned and shown in listings 2.2 and 2.3. Only what we need is to replace the UNIQUE keyword with the PRIMARY KEY keyword.

FOREIGN KEY integrity constraint

A FOREIGN KEY integrity constraint is an existing value from related table where its primary key is stored in a certain row and the value of foreign key is stored in another row(s) in the another table. There is a relation between two tables where the first table must have its primary key values and the other one table must contain values of the primary key as values of the foreign key,

but if it is necessary. We say this maintains the referential integrity between two related tables [32]. Or we can follow an Oracle definition as the best way [33]: "Foreign keys provide a way to enforce the referential integrity of a database. A foreign key is a column or group of columns within a table that references a key in some other table."

Foreign keys are also possible to define in the both levels at a column and table level. Listings 2.4 and 2.5 shows us how to create reference between the PRIMARY KEY and the FOREIGN KEY.

Listing 2.4: PostgreSQL PRIMARY KEY constraint as a column definition [32]

```
CREATE TABLE products (
        product_no integer PRIMARY KEY,
        name text,
        price numeric
);
```

Listing 2.5: PostgreSQL FOREIGN KEY constraint as a column definition [32]

```
CREATE TABLE orders (
        order_id integer PRIMARY KEY,
        product_no integer REFERENCES products (
           product_no),
        quantity integer
);
```

First of all we need to define a new table *products* with a primary key; secondly we need to create another table *orders* which will use the primary key from the first table as a foreign key in the second table. This is done via a keyword called REFERENCES.

However, there is a problem which the other integrity constraints do not have and it is how we will react if we want to apply DML operations which include delete and update operations. Both databases Oracle and PostgreSQL have the same solution. They use referential actions where we can specify an *ON DELETE clause* and an *ON UPDATE clause* which are followed by the appropriate action [33]:

- NO ACTION,
- SET NULL,
- RESTRICT, or
- CASCADE.

The next small part introduces how actions can affect the ON DELETE clause when it is raised and a few differences between both databases Oracle and PostgreSQL database.

NO ACTION

An action NO ACTION in PostgreSQL means raising an error message if there are any referencing rows into table(s), when the integrity constraint is checked [32]. This action in PostgreSQL is set by default. Oracle database executes update or delete operation and then checks the dependent tables for foreign key integrity constraints. If there is a value which breaks the integrity constraint rule, then the statement is rejected. Oracle database has set by default NO ACTION too.

SET NULL

A SET NULL action is same for both databases, but Oracle has the SET NULL clause only for a delete operation [33]. The SET NULL possibility writes a null value to a child table when a referenced row in a parent table is updated or deleted, for Oracle database this statement is true only for the delete operation. For the delete operation a record in the parent table is deleted but the child table will not be affected and its value will be set as the null value. On the other hand PostgreSQL has in addition a SET DEFAULT action which means, apply the same statement like for the SET NULL, but instead of the SET NULL clause we will use the default value which were set at the begging of the integrity constraint definition [32].

RESTRICT

Generally, a value cannot be updated or deleted if we have an existing row in a base table with a reference value in a referenced table [34]. A RESTRICT action in PostgreSQL prevents from delete or update operation in the referenced row. The RESTRICT rules are checked before any other operation [35]. Oracle database does the same. Firstly, it checks dependent tables for foreign key constraints, and then if any row in a dependent table does not fulfill a foreign key constraint, a transaction is rolled back [33]. Clearly, the RESTRICT action checks the integrity constraint before the execution of the update or delete statement.

CASCADE

A CASCADE action is also applicable in PostgreSQL for both operations update and delete but Oracle database. Oracle only supports the CASCADE action for the delete operation. A foreign key with a cascade delete action means that if a record in a base table is deleted, then the records in child tables will be deleted automatically too [36]. This rule is corresponding with the update operation in PostgreSQL where changed value in the base table will be automatically propagated to the child tables.

After a short brief of actions that can affect tables on delete clause in our database we can take a look at listing 2.6 where is an example covering the usage of these operations in practice by PostgreSQL database.

Listing 2.6: PostgreSQL FOREIGN KEY constraints, ON DELETE clauses [32]

```
CREATE TABLE products (
        product_no integer PRIMARY KEY,
        name text,
        price numeric
);

CREATE TABLE orders (
        order_id integer PRIMARY KEY,
        shipping_address text,
        ...
);

CREATE TABLE order_items (
        product_no integer REFERENCES products
                                ON DELETE RESTRICT,
        order_id integer REFERENCES orders
                                ON DELETE CASCADE,
        quantity integer,
        PRIMARY KEY (product_no, order_id)
);
```

CHECK

A CHECK integrity constraint is good if we want to specify a requirement, which will control rules set up for table columns. On background runs a search condition which is a Boolean expression and the search condition must be satisfied for all rows in the column or table where it is defined. The CHECK integrity constraints are used with update and insert operations. The whole statement is aborted if any check integrity constraint is violated [33]. The CHECK integrity constraint also can be used for multiple columns. Listing 2.7 shows us how to use this constraint constraint.

Listing 2.7: PostgreSQL CHECK constraint [32]

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric CHECK (price > 0)
);
```

Managing Integrity Constraints

In some cases we can have a database where we have defined too many integrity constraints. In that particular case we definitely need a kind of a special tool which helps us to handle these integrity constraints. Integrity constraints are strict rules watching inserted values into one or more columns in a table. In Oracle database integrity constraints can be defined in two ways. First one is a CREATE TABLE and the second one is an ALTER TABLE statement. In these statements we identify the column or columns and assign a certain integrity constraint condition [37, 38].

If we want to know information how many defined integrity constraints do we have, we can find out that this possibility is only available in Oracle database because PostgreSQL database has not implemented needed functionality yet. Oracle database for an integrity constraint management provides views where it is possible to query them with a SELECT clause. These views are named DBA_CONSTRAINTS or its alternative ALL_CONSTRAINTS which describe all integrity constraint definitions in the database, or integrity constraint definitions accessible to current user, respectively. The another one views are called as DBA_CONS_COLUMNS or ALL_CONS_COLUMNS which describe all columns used in the database that are mentioned in integrity constraints or similarly all columns appeared in the integrity constraint specification under the current user [38].

According to Oracle, it is specified that an integrity constraint can be either states *ENABLE* or *DISABLE*. These two states have its own meaning. If the integrity constraint is passed as ENABLE, then data are checked whenever they are updated or inserted on the database, and data that do not fulfill the integrity constraint condition are not stored into the database. If the integrity constraint is passed as DISABLE, then data that do not conform the integrity constraint condition can be allowed to store themselves into the database [38].

In addition, in Oracle database we can specify more detailed integrity constraint condition. It is done by *VALIDATE* and *NOVALIDATE* keywords. The VALIDATE means that existing data in a table must satisfy the integrity constraint. The NOVALIDATE means that we are not ensured whether our existing data conforms the integrity constraint [38]. Due to those four states we are able to gather them and we get following paired phrases:

- ENABLE VALIDATE,
- ENABLE NOVALIDATE,
- DISABLE VALIDATE,
- DISABLE NOVALIDATE.

For those four states could appear one question. Why should we want to use disabling and enabling of integrity constraints? The solution is ob-

vious if we had an ability to disable integrity constraints in some cases, we would be able to manipulate our database from these following performance reasons [38]:

- When we load large amount of data into a table.
- When we call operations that make huge changes in a table.
- When we import or export one table at a time.

And there is a possible way to enter particular data which violate integrity constraint condition in a database if we for a small period of time disable the specific integrity constraint(s). After performed all operations the integrity constraints should be activated.

Enable validate

A relation ENABLE VALIDATE is an action which starts automatically after defining an integrity constraint. Typically a CREATE TABLE statement, or if we specify it in an ALTER TABLE statement. This part of integrity constraint starts immediately validating of the whole database. During the integrity constraint definition the specification of VALIDATE action is not needed. The behavior between actions ENABLE VALIDATE and only ENABLE is the same. There exists one difficulty and it is if we want the database validation during data loading. For example we have an empty database and we want to fill it with data. If the integrity constraint is enabled, then everything else will be extremely slowed because any included data will have to be validated which is really time consuming operation [38, 39].

Enable novalidate

An ENABLE NOVALIDATE phrase is a little bit different than the ENABLE VALIDATE phrase. This ENABLE NOVALIDATE is a missing part which is really useful in a way where present data in a database cannot be validated. Due to this possibility, for example, we are able to disable the integrity constraint validation then apply some huge data changes or just simply copy tables where data will not be validated because time consumption and after all needed operations will have been done, then we enable the integrity constraint with the NOVALIDATE action. Enabled constraint with NOVALIDATE action is applicable only for newly inserted data or newly updated data. There is a risk in disabling validation mode because it could appear data that violate the integrity constraints [38, 39].

Disable validate and disable novalidate

These two phrases are designed to disable integrity constraints. There is one difference between them and it is that DISABLE VALIDATE construct disallows the integrity constraint and its managed index, which were created on that integrity constraint, will be dropped. The DISABLE NOVALIDATE behavior is the same as if we used only the DISABLE action [40].

DROP constraint

It could happen that some defined integrity constraints are no more useful at all. If the rule is no longer truthful needed to enforce the integrity constraint, we can easily drop the integrity constraint by ALTER TABLE statement. This technique is the same for both databases Oracle and PostgreSQL. Listing 2.8 shows us a manner how to use the DROP construct with the ALTER TABLE command. We can see that the syntax for dropping integrity constraint by the ALTER TABLE is preserved and it is the same as the other used clauses.

Listing 2.8: PostgreSQL DROP constraint [41]

```
ALTER TABLE table_name (
        DROP CONSTRAINT constraint_name
);
```

DEFERRED or IMMEDIATE constraints

This functionality can be found in both databases Oracle and PostgreSQL. It is a useful thing if we want to manipulate or postpone an integrity constraint validation process. There are two possibilities [42]:

- DEFERRABLE INITIALLY IMMEDIATE or
- DEFERRABLE INITIALLY DEFERRED.

By default, with no mention in the integrity constraint construct is a behavior automatically set as NOT DEFERRABLE which means that integrity constraint is checked immediately [43]. These options can be controlled with a *SET CONSTRANTS* command, but it has one difficulty. If we have an earlier defined integrity constraint which does not contain the DEFFERABLE formula, then it is a must to drop that integrity constraint and re-create it again with the DEFFERABLE construct.

However, is that the main difference in using the DEFERRABLE construct? If we define an integrity constraint with no DEFERRABLE keyword, we will obtain behavior in a form where on every insert, or update operation the integrity constraint will be called. This is a general behavior. But, if we define a new integrity constraint which would contain a DEFERRABLE keyword

plus a part behind, the INITIALLY IMMEDIATE, or DEFERRED clause, then we will be able to handle it via SET CONSTRAINTS command if needed which is the added value.

A defined DEFERRABLE INITIALLY IMMEDIATE construct for integrity constraints will have the same behavior as the default, i.e. NOT DEFERRABLE. The meaning is that integrity constraints in the current transaction are checked at the end of the statement. Using of DEFERRABLE INITIALLY DEFERRED construct means that integrity constraint assertion is not checked until a transaction COMMIT is called. After the COMMIT phase is run the validation of particular integrity constraint. If there are some violences, the performed actions will be rolled back[44].

Oracle database supports this extension for all kinds of integrity constraints and PostgreSQL almost too, except for the NOT NULL and CHECK integrity constraints which are always checked immediately with insert, or update operations [42].

Listing 2.9 shows an integrity constraint definition with no mention about DEFERRABLE keyword, but meaning and behavior is NOT DEFERRABLE thus like IMMEDIATE. Otherwise listing 2.10 shows the same integrity constraint, but using the DEFFERABLE formula which can be later handled by SET CONSTRAINTS commands.

Listing 2.9: Oracle CHECK constraint, NOT DEFERRABLE [45]

```
CREATE TABLE games (scores NUMBER CHECK (scores >= 0));
```

Listing 2.10: Oracle CHECK constraint, DEFERRABLE included [45]

```
CREATE TABLE games (
        scores NUMBER,
        CONSTRAINT unq_num UNIQUE (scores)
        INITIALLY DEFERRED DEFERRABLE
);
```

Of course, there are some differences in grammar syntax in both databases Oracle and PostgreSQL, but behavior is the same. A SET CONSTRAINTS construct for Oracle is defined as (2.11):

Listing 2.11: Oracle SET DEFFERED [45]

```
ALTER SESSION SET CONSTRAINTS = DEFERRED;
```

and in PostgreSQL as (2.12):

Listing 2.12: PostgreSQL SET DEFFERED [42]

```
SET CONSTRAINTS { ALL | name [, ...] }
                { DEFERRED | IMMEDIATE }
```

Id	Position	Name
25	1	Bread
68	2	Milk
72	3	Eggs

Table 2.1: Shopping list before changing the position [43].

Id	Position	Name
25	1	Bread
68	2	Milk
72	1	Eggs

Table 2.2: Shopping list changing the position [43].

Table 2.1 and 2.2 shows us how different are DEFFERED and IMMEDIATE clauses. Let explain it on a creation of a unique integrity constraint at a column named position. So, there should not be possible to have the same position number twice at the time at the position column in the table. Our task demonstrates an update operation where we want to shift an Eggs item on our shopping list and mark it to the higher position, for example, mark with one. If we left a default option which is validating the integrity constraint immediately, hence IMMEDIATE, we will obtain the integrity constraint error message that we are not allowed to change a position due to the integrity constraint violation.

This is being done because the database executes every statement (here an update operation) immediately. We have already known before that if we change one position in the shopping list, we will have to change it to the another position which violates the integrity constraint rule, but the database does not know it at that time. But, there is a solution in a DEFERRED clause. The DEFERRED clause was designed to avoid this conflict. The database lets us to make changes, thus the change of position for Eggs to one and also change position for Bread to e.g. four, and then after we explicitly COMMIT the changes, the integrity constraint will check for collisions. There should not appear any collisions because we did the things right.

2.2 Cypher Schema syntax – Research of integrity constraint proposals for Neo4j

This section contains taken and re-formulated proposals from a *Request for proposals: Cypher Schema syntax* paper published by a representative of Cypher

Language Group Tobias Lindaaker [46]. The RFP are in most cases inspired from integrity constraints used in the relational databases which were mentioned in section above 2.1. This research introduces and describes what kind of integrity constraints should be defined in Neo4j graph database. Especially, it is explored a following range of integrity constraints:

- Node property uniqueness,
- Mandatory properties,
- Property value limitations,
- Required relationships,
- Cardinality requirements,
- Endpoint requirements,
- Label coexistence.

Node property uniqueness – Uniqueness constraint

This kind of integrity constraint specifies a certain limit among nodes. Each node should have given one label, its identifier. For every node is applied to have a certain attribute, called a property key, with a value. Thus nodes store values in a key-value combination, there must exists an option for the node that has the particular value for the given property key, or the particular combination of values for the particular set of property keys.

Examples:

- Among all nodes with a specified label as **User**, each node must have a unique value for the properties like *email, username*, etc.
- Among all nodes with a specified label as **Person**, each node must have a unique value combination for the *firstName* and *lastName*. Possible combinations for two nodes that could be fine are:
  - Both nodes can have the same *firstName*, but no *lastName*, or
  - Both nodes can have the same *lastName*, but no *firstName*.

Mandatory property constraints

Mandatory property constraints should be applied for both the nodes and relationships. This integrity constraint specifies for a node with a particular label that must have assigned a value for a given property. The same holds true for a relationship with a particular type. The lack of mandatory property

should not allow creating a new node, or a relationship.

Examples:

- Each node with a specified label **User** must have filled a value for a mandatory property *email*.
- Each relationship with a type **ROAD** must have filled a value for a mandatory property *distance*.

Property value limitations

The next integrity constraint is called Property value limitations which determines for a particular label of a node, or a particular type of a relationship, if a particular property has a value then that value must meet the criteria to the given specified rules.

Examples:

- Each relationship with a type **ROAD** and a *distance* property must be finite number greater than 0.
- Each node with a specified label **User** and an *email* property must be of type **String** that must match a regular expression "$[a-z0-9](([a-z0-9\-.]+[a-z0-9])?@([a-z0-9]([a-z0-9\-.]+[a-z0-9])?\.)+[a-z]+$".
- Each node with a specified label **User** and an *active* property must be of a **Boolean** type.
- Each node with a specified label **Vehicle** and the *locations* property must be a list of geospatial points.

Required relationships

The Required relationships integrity constraint defines for a certain node with a particular label that must have one or more relationships with a particular type.

Examples:

- Each node with a specified label **Person** must have an outgoing relationship of type **LIVES_AT**.
- Each node with a specified label **Sink** must have an incoming relationship of type **FLOW**.
- Each node with a specified label **Place** must have a relationship of type **NEIGHBOUR**, in any direction.

Cardinality requirements

The Cardinality requirements integrity constraint is closely related to the **Required relationships integrity constraint** 2.2, but instead of requiring a relationship with a certain type to a particular node, this integrity constraint specifies the cardinality. The cardinality is the minimum and maximum number of relationships with a certain type that a given node with a particular label must have contain.

Examples:

- Each node with a specified label **SwedishCitizen** may have at most one relationship of type **MARRIED_TO**, in any direction.
- Each node with a specified label **Person** must have exactly two incoming relationships of type **PARENT_OF**.
- Each node with a specified label **BuddhistMonk** may not have more than 14 outgoing relationships of type **OWNS**.

Endpoint requirements

This kind of integrity constraint specifies a relationship with a particular type that must, or must not, start or end in the certain nodes with a particular label, or sets of lables.

Examples:

- Each relationship with a type **OWNS** must start at a node with either a Person label or an **Organisation** label.
- Each relationship with a type **OWNS** must end at a node with either a **Vehicle, Building, Item** label or an **Organisation** label.
- If each relationship with a type **OWNS** ends at a node with an **Organisation** label, then it must start at a node with a **Person** label.
- Each relationship with a type **WORKS_FOR** must start at a node with an **Employee** label and must end at a node with both the **Employee** label and **Manager** label.
- Each relationship with a type **OWNS** must not end at a node with a **Person** label.

Label coexistence

The next and the last integrity constraint is Label coexistence. The label coexistence integrity constraint specifies that two certain node labels may not occur on the same node, or that a particular label may only occur on nodes with a different particular label.

Examples:

- Each node may not have both labels a **Person** label and an **Organisation** label.
- Each node may only have the **User** label in the case if it is has also the **Person** label.

2.3 Cypher Schema syntax – Revision of integrity constraints

In Neo4j integrity constraint analysis section 2.2 we showed the integrity constraint proposals introduced by a representative of the Cypher Language Group Tobias Lindaaker. The mentioned integrity constraint design predominantly comes from those integrity constraints which are implemented and used in the relational databases. However, for those integrity constraints taken from the relational world some things were not discussed and this is the right section where we should deal with it. With help of our research dealt on PostgreSQL and Oracle databases, elaborated in section 2.1 we present possible issues that must to be ensured. We will take listed problems into account, theorize them and show from a great point of view how they can affect an eventual process of integrity constraint validation in Neo4j graph database.

In general, figure 2.2 shows us a range sphere of interest for each integrity constraint and how we proceed further. At the forefront it is laid the integrity constraint itself. Then follow the rectangles where it is important how large they are. Simply, the larger rectangle is, the more often is mentioned in a text. Figure 2.2 covers all possibilities which represent the knowledge we acquired before. This includes time *Complexity* when statements are executed, this involves both states empty and full database. The next is a validation process separated into two parts, *DML operations* and *Time execution* where is very important to describe *behavior* that can affect the graph database. As a reminder, DML operations accompany **Behavior** where belong clauses like **NO ACTION, SET NULL, RESTRICT and CASCADE** discussed in chapter 2.1. This can be useful in discussion with problem alike foreign key constraint. The next parts are *Discussion* and *Usefulness* where appropriate explanation will have been appeared and the last one is a *Syntax* which is not much

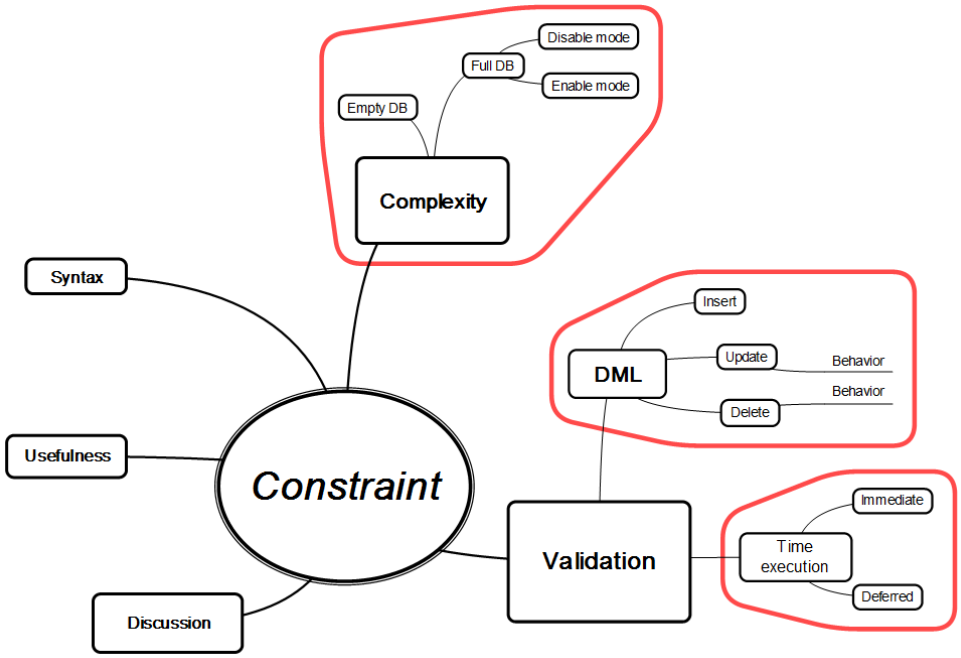


Figure 2.2: Mind map constraint issues.

mentioned in this section, but in the next chapter 3 **Cypher syntax design** dealing with a grammar syntax for Cypher.

Node property uniqueness

This kind of integrity constraint, in a certain limited form, in Neo4j graph database already exists. However, Cypher Language Group on the bases of their analysis of new potential integrity constraints decided to re-formulate the unique integrity constraint into new form for sure. It was intended this way because if it is proposed the new Cypher syntax for defining the integrity constraints, then this integrity constraint has possibly to be entered by the new Cypher syntax too.

The actual Cypher syntax for the Node property uniqueness integrity constraint is shown in listing 2.13:

Listing 2.13: Node property uniqueness - actual Cypher syntax

```
(1) CREATE constraint ON (p:Person) ASSERT p.Name
                                            IS UNIQUE;
```

The disadvantage of the current integrity constraint definition is that it is not possible to assert more properties simultaneously, respectively there does not exist the unique integrity constraint covering multiple properties

simultaneously. To make this thing possible was needed to use a small hack, which consequently validates the uniqueness for the multiple properties. The solution was based on the form of string concatenation.

The new proposal already takes this into account. It is intended to be possible for each node which has specified a particular label to assert a unique value for multiple properties simultaneously. The solution was mentioned in analysis chapter, research section 2.2 where for each node with a label *Person* is possible to assert uniqueness for both properties the *firstName* and the *lastName* simultaneously. If we take a look at **Usefullness** the usefulness is more than obvious. We need data to be stored in the database which must not repeat under the certain property and this is the major task of this integrity constraint, to watch the property uniqueness.

Nevertheless the most important thing is mainly behavior from two points of view, the **Complexity** and **Validation**. These two parts are very closely related. The Complexity describes a time spent in a validation process which is performing the database check (or an integrity constraint validation) control. The **Validation** description is important during performing cost-expensive operations such as when to execute the integrity constraint validation if we have large amount of data in the database or how will we act if we load large amount of data, etc.

The unique property validation process has the same impact for both operations *insert* and *update* and also for both database state which can be a full database or an empty database. The validation process must be executed across all the property values that are stored in the database under the specified certain node label. This leads to *O(n)* complexity where *n* is a total number of nodes in the database (without optimization) or with an optimization *n* can be a number of nodes with a given particular label, but with no database overhead. For the optimization could be chosen some kind of index implementation. The *delete* operation has *O(1)* complexity and it is given by the influence of Neo4j architecture mentioned in section 1.2. We are able for each node by his internal identification number (i.e. ID) to compute and find the exact physical node position. Then, we can just enter a value into *in-Use* flag to mark a specific node as an inactive node.

However, if we decide for a possibility where we want to take into consideration enabling and disabling integrity constraint in the database, it is needed to introduce a few countermeasures. This use case can happen if we have large set of data and we need the data to be loaded into the database. If we had turn on the integrity constraint validation for checking unique values, the all loading operations would last too long. Therefore, we can draw inspiration from the world of relational databases where we mentioned in PostgreSQL and Oracle analysis 2.1 these possible actions where belong keywords like *enable validate* or *novalidate* and *disable validate* or *novalidate*. For our purpose we only need if we were able to disable integrity constraint, and then would enable it under those two options *enable validate* or *novalidate*. Let

Property values	
A	D
B	E
C	F

Table 2.3: Property values of given nodes.

go back to our example where we have a big package of data to be uploaded to the graph database. In the case that we are 100 percent sure that the data are consistent with the desired integrity constraint rule, we can perform a sequence of following operations: *disable integrity constraint* → *load data* → *enable integrity constraint with novalidate keyword*.

In section 2.1 we mentioned that *enable novalidate* is applicable only for newly inserted data or newly updated data. With this capability the overhead is eliminated that would have to do the proper validation under the loaded data. The validation control process would take more time if the integrity constraint was turned on as *validate*. But, if we are certain of data correctness, it is not necessary to perform the integrity constraint validation. The disadvantage of this step is that if the data are not valid with respect to the defined integrity constraint, it can lead to data inconsistency in the database, but this decision is on our responsibility. If we want to avoid inconsistencies, we should follow this flow of operations: *disable integrity constraint* → *load data* → *enable integrity constraint with validate keyword*.

In case if we are unsure of data correctness, then the turning on the *enable validate* control will be checked the whole database under the uniqueness integrity constraint. Shortly afterwards is necessary to check each property value to the other and this leads in the worst case to $O(n^2)$ complexity. The reason why this is $O(n^2)$, is as follows. Imagine nodes with a specified label and a given property which we must validate its value. Table 2.3 represents for each node one table cell illustrating a value of a certain property. Totally we have given n nodes, i.e. $n = 6$.

To validate the unique integrity constraint we start at a label with a property value **A**. This property value will not be controlled, but we need to check the other $(n - 1)$ positions with the omitted value **A**. Furthermore, the same applied for the property value **B, C** and others where also is needed to be checked $(n - 1)$ positions. This leads to $O((n - 1)(n - 1))$ complexity which is $O(n^2)$ in the worst case. In case the validation of the property combinations and its values the time complexity will not be so noticeable. In other words the combination of values has not a marked effect on the time complexity.

Unfortunately, this kind of integrity constraint disrupts a user modification process. Especially, in the certain cases, primarily between editing where we need, for example, switch a few values, but in time of changing the val-

ues are violating the integrity constraint rule, like it is demonstrated on the shopping list example in section 2.1. In certain contexts when we have the integrity constraint turned on may occur during the *update* operation to adverse effects. The solution is taken from the relational databases where they were sorted this out with using of a *deferred* statement execution. The data validation process is performed after calling a *commit* command. In practice, it would look as follows. The integrity constraint would have been set to deferred. This state allows us to perform a series of *update* operations, but if we had had at that time the integrity constraint enabled, the *update* operations would have been violated and stopped making changes. Thanks to deferred option this would not happen and therefore the necessary updates can be performed, and after the explicit *commit* would start the database validation process.

Mandatory properties

The expected behavior of the *Mandatory property constraints* we can also meet to a lesser extent in Neo4j database. It is an existential integrity constraint which was mentioned in a section dealing with the integrity constraints in Neo4j 1.2. This existential integrity constraint is used to enforce a property existence and its value. We have indicated that this option is only available in the *Enterprise Edition* of Neo4j graph database. The existential integrity constraint supports asserting properties for both nodes and relationships. Like in the previous section dealing with *Node property uniqueness* integrity constraint this existential integrity constraint may be included among candidates which can be replaced with a new Cypher syntax. The actual Cypher syntax for mandatory properties for both nodes and relationships is shown in listing 2.14.

Listing 2.14: Mandatory properties - actual Cypher syntax [23]

```
(1) CREATE CONSTRAINT ON (book:Book) ASSERT
                                    exists(book.isbn)

(2) CREATE CONSTRAINT ON ()-[like:LIKED]-() ASSERT
                                    exists(like.day)
```

The first Cypher statement **(1)** tells us that the integrity constraint is applied to the all nodes with a particular label *Book*. Every node with the *Book* label requires to have included a property *isbn*. This *isbn* property must be NOT NULL. The second statement **(2)** defines a data validation rule for relationships where each relationship that has a type *LIKED* must enforce a *day* property as a mandatory property that should not be absent.

The *Mandatory property* integrity constraint enforces the existence property that can be considered as a NOT NULL integrity constraint which is mostly

known from the relational databases where it is also possible to have the null values to be inserted into the table cells. In the graph database case the null values into the properties cannot be stored. The null value is represented by the property non-existence. For example if for some property is said to be null, it is useless to establish the property in the graph database. On the other hand, if some property must exist (i.e. must be NOT NULL), we will enforce it with the mandatory integrity constraint.

The answer in terms of time complexity for the *insert* operation, thus creating nodes and relationships is clear. The integrity constraint control will always take place locally on the statement which creates a new element (i.e. node or relationship) and the check itself will last $O(1)$ complexity. The same applies for the *update* and *delete* operations. The update operation is performed in Cypher syntax by combination of MATCH and SET clauses where it is able to find the proper results and those results after performed changes can be also validated in the constant time. But, the main thing we should have had considered before is behavior for the *delete* operation. It may be a case that we would like to delete a property which had earlier defined an integrity constraint to watch that property. Respectively, we could mention as an example keeping records for books. The node label would be named as a *Book*, next we would have properties for books like *name, release date* and *isbn*. Note, it is not a good idea to be inserted as a property the author of the book into the *Book* node. The database schema should be suitably designed for proper querying and it is good to have *Authors* as the separate nodes. The joining authors with books would be managed, for example, through the relationship type *WRITTEN*. However, in the case where we have the integrity constraint turned on and we would performed the delete operation (listing 2.15) for remove property which the integrity constraint is actually validating, it should be expected that the defined integrity constraint should raise an exception and should not allow us to delete the certain property (here *isbn*) because due to the existential integrity constraint, the property must exist in the database (i.e. must be NOT NULL).

Listing 2.15: Mandatory properties - delete - actual Cypher syntax

```
(1) MATCH (b:Book) REMOVE b.isbn return b
```

The same discussion that took place as the *Node property uniqueness* constraint 2.3 can be also interested in the issue of behavior when there are loaded large amount of data into the database machine with our aim which is to avoid the adequate slowdown during loading data into the database. Again, we can suggest the same proposal as above which is the use of mentioned two options *enable* and *disable* where the enable is devided into cases the *enable validate* or *novalidate*. The only difference is in the fact that if we turn on after database data loading the integrity constraint with *enable validate* clause, it will have to be validated all the inserted nodes and relationships since the

time the database was firstly initiated. The validation process asymptotically takes $O(n)$ in time complexity. On the other hand if we enable the integrity constraint with *enable novalidate*, then the previously entered data will not be checked and validated under the present integrity constraint because by the casting the *novalidate* clause we guarantee the data correctness. The whole data validation process will be carried out for new incoming data or during updating the values under the existing properties under the enabled integrity constraint. We do not take care with *DEFERRED* and *IMMEDIATE* clauses due to its inapplicability in this integrity constraint use case.

Property value limitations

The next integrity constraint to revise is *Property value limitations*. This is the first integrity constraint which does not have an old Cypher syntax and also like the next other integrity constraints including *Property value limitations* currently do not exist in Neo4j graph database. The *Property value limitations* integrity constraint is a new proposal which is considered its application in Neo4j. This integrity constraint finds its value in several ways. As the integrity constraint name implies, the validation process is related with some limits ruled on the certain properties. This integrity constraint is designed to control properties for both nodes and relationships.

In research section 2.2 are shown examples for the property value validation, and how such the integrity constraint might work in practice. In listing 2.16 is exposed a pseudo Cypher syntax for the few specified examples in 2.2.

Listing 2.16: Property value limitations - pseudo Cypher syntax

```
(1) CREATE CONSTRAINT ON (book:Book) ASSERT
                                    STRING(book.isbn)

(2) CREATE CONSTRAINT ON ()-[road:ROAD]-() ASSERT
                              EXISTS(road.distance > 0)

(3) CREATE constraint ON (user:User) ASSERT user.email
    AS REGEX("[a-z0-9]([a-z0-9\-.]+[a-z0-9])?@
          ([a-z0-9]([a-z0-9\-.]+[a-z0-9])?\.)+[a-z]+");

(4) CREATE CONSTRAINT ON (v:Vehicle) ASSERT
                    EXISTS(List<Points> for v.locations)
```

The first example shows us a possibility of choice what type of data type we would like to have to keep in a property with a specific node labeled as a *Book*. There is no specification what kind of data types should be supported except for the *Boolean* type. However it would be possible for the *Property value limitations* integrity constraint supports those values that are described in

the Java Language Specification. The next example deals with a relationship property. The relationship of type **ROAD** and a *distance* property defines a particular restriction in a set of integers $\mathbb{Z}$ where we could accurate a specification with use of mathematical symbols like "<, >, =" and also their combinations from which interval would be allowed to take values. The other example shows property validation via a regular expression and in the last example is shown enforcing an array of points in a form of a list datatype.

The enforcement of this integrity constraint in terms of complexity is very similar to the integrity constraint dealing with the mandatory properties mentioned above. When inserting new data to the database with a CREATE statement a validation process will be performed immediately after commit and the validation itself will be done in a constant time. The same holds true for updating where the data validation will also take the constant time. In terms of time complexity is no need to mention a *delete* operation because it does not depend on the behavior of the integrity constraint definition, but the whole operation also executes at $O(1)$ complexity.

When comes a question regarding to update or load of large amount of data to the database, for this serves a temporary disabling the integrity constraint for speeding up such operations. We talk about the same use case mentioned in the previous integrity constraint discussion. If we have a huge package of data that needs to be loaded into the database, we will certainly want to avoid the time delay which would be occurred in the case of the enabled integrity constraint rule. We are offered an option to use a solution from the relational databases where it is followed in these steps *disable integrity constraint* → *load data* → *enable integrity constraint with validate or novalidate keyword*. The other possibility is if the amount of data is small and we would not want to disable the integrity constraint, then we could take an advantage in postponed time validation which means that the integrity constraint must be defined as **DEFERRED**. In this situation the statements would be performed and after the explicit commit the loaded or updated data would be checked whether they do not violate to the specific integrity constraint rule.

Required relationships

The aim of the integrity constraint named as *Required relationships* is to define such integrity restriction which will control the relationships required by the particular nodes with a specific label. Research in section 2.2 indicates that should be allowed the verification of three direction types:

- Outgoing,
- Incoming, and
- In any direction.

Under the influence of preserving the current definition of the integrity constraint syntax for Neo4j, for now, we can draw an inspiration from the pseudo Cypher syntax where the examples from section 2.2 are clearly explained in listing 2.17.

Listing 2.17: Required relationships - pseudo Cypher syntax

```
(1) CREATE CONSTRAINT ON (person:Person) ASSERT
                                ORELATIONSHIP(LIVES_AT)
(2) CREATE CONSTRAINT ON (sink:Sink) ASSERT
                               IRELATIONSHIP(FLOW)
(3) CREATE CONSTRAINT ON (place:Place) ASSERT
                                RELATIONSHIP(NEIGHBOUR)
```

In the Cypher pseudo-code syntax are directions determined in this way. The first example (line number **(1)**) is needed to be validated an outgoing relationship with type *LIVES_AT*. This kind of relationship could be recognized with the first letter added to the *relationship* clause under which must be performed the validation process, this first letter would be **O** as *outgoing* and the combination gives a result as *ORELATIONSHIP* clause. The same principle could be applied if we consider an *incoming* relationship where insted of the letter **O**, we will use the latter **I** and we get the *IRELATIONSHIP* clause. The third example (line number **(3)**) tells us a possibility of the integrity constraint assertion for relationships in any direction. For the last use case defining the validation could be used the *RELATIONSHIP* statement itself.

What the *Required relationships* proposal lacks is any discussion how should look like the appropriate implementation, thus, what are the technical issues resulting from this integrity constraint. Because we have to take into account a time execution in which the *Required relationships* should be activated. By this reasoning we can try to avoid too early integrity constraint violations before the time the data are loaded in the database. For this use case could be handy one solution that is implemented in the relational databases. The solution can be called as a *deferred validation*. Here is an example describing the problem. We want to manually load data into the database and we have decided to load nodes and then that nodes join with particular relationships. Unhappily, if the integrity constraint did not have enabled the *deferred* mode, immediately after the creation of nodes, would inserted data violate the integrity constraint condition. Simply, the execution time in the integrity constraint enabled as *immediate* violates the database schema which the integrity constraint set and restricted before. Therefore there must be considered an option of using the *deferred* keyword served to make possible the validation process to be postponed.

Regarding the loading and updating problems with large amount of data are suitable like in the previous cases. As a sequence of these steps could be used *disable integrity constraint* → *load or update data* → *enable integrity constraint*

with validate or novalidate keyword. If we used an *enable validate* option, then the asymptotic complexity would be $O(n)+O(1)$ where n is a total number of data stored in the database and $O(1)$ is a constant time for the validation process. The explanation is as follows. After performed data load action we enable the integrity constraint as *enable validate*. The actual graph is extended about new nodes and relationships, and with that activated integrity constraint there is needed to check the whole database and thus any data which has mentioned the restriction in the integrity constraint. The verification process checks the node after node which is $O(n)$ complexity and during that time by using micro-indices we also verify the node neighbors which is $O(1)$ complexity. Totally, we have for the validation loaded or updated data $O(n)$ complexity. The same $O(1)$ complexity applies for *insert*, *update* and *delete* operations. It is based on a fact that each node is designed to behave like a micro-index and is able to look after his siblings. More information about micro-indices can be found in a Neo4j architecture section 1.2.

Cardinality requirements

The *Cardinality requierements* integrity constraint is very closely associated with the previous integrity constraint *Required relationships*. This integrity constraint should be useful in fulfilling its core task and that is in watching node cardinality validation. Under this concept we can imagine a defined restriction imposed on a number of relationships that a certain node can have mapped. For demonstration are used examples from the research section in 2.2. Because of a similarity with the previous topic we remain in the use of equivalent pseudo Cypher syntax where listing 2.18 illustrates the proposed examples.

Listing 2.18: Cardinality requirements - pseudo Cypher syntax

```
(1) CREATE CONSTRAINT ON (sc:SwedishCitizen) ASSERT
                          RELATIONSHIP(MARRIED_TO <= 1)

(2) CREATE CONSTRAINT ON (person:Person) ASSERT
                          IRELATIONSHIP(PARENT_OF == 2)

(3) CREATE CONSTRAINT ON (bm:BuddhistMonk) ASSERT
                         ORELATIONSHIP(OWNS <= 14)
```

The first example listed as **(1)** tells us that we want to define an integrity constraint that sets a rule for a node labeled as *SwedishCitizen* that may only have at most one type of relationship MARRIED_TO. The relationship can acquire both directions the incoming or outgoing. This relation in a pseudo Cypher notation is interpreted by a combination of mathematical symbols "<=" and a numeral one. The second example **(2)** shows that a node with

a specific label *Person* must have exactly two incoming PARENT_OF relationships. The Cypher pseudo syntax was inspired from the example **(2)** in listing 2.17. Thus, using the first letter **I** as an *Incoming* indication which then the letter is inserted before the relationship keyword and we obtain the *IRELATIONSHIP* clause. In pseudo Cypher syntax the IRELATIONSHIP takes defined rules that must be checked. This rule uses mathematical symbols "==" with a specified numeral, here number 2. The third and last example defines the integrity constraint for a node labeled as *BuddhistMonk* where is a restriction in a form of limiting the total number of the relationships with a type OWN. This kind of relationship is *Outgoing* and with the use of the same principle with incoming relationship we obtain *ORELATIONSHIP* clause ready to enter the enforcement rule.

Loading or updating the large amount of data is confronted with the same problem as the previous all integrity constraints. There must be considered the possibility of the temporary disabling of integrity constraint. As the next would be followed for us known steps in the form of integrity constraint disabling, then performing the necessary actions, and then repeated constraint enabling with validate, or novalidate keyword. This is the still repetitive case and there is no need to discuss here any more.

The next thing what is needed to mention at this integrity constraint is to have an ability to defer a validation time execution because we should be able to avoid the integrity constraint validation process in the case if the data are not fully loaded, or updated in the graph database. As a recommendation we should have defined this integrity constraint as a*deferred*, not an *immediate*, with a combination of explicit commit.

The complexity for *insert* and *update* operations should not be great. If we focus on the individual nodes and their siblings, we talk about a constant time because of the Neo4j feature called as a micro-index. This advantage consists in the fact that an adjacent node can be reached in one step which is $O(1)$ complexity. What is needed to be fully processed are obligations of cardinality of the integrity constraint. It is of particular importance, the behavior that evokes the obligation in the form of mathematical symbols such as "==, <= or <". For both operations *insert* and *update* are needed to utilize the late check integrity constraint functionality. Further, if we concentrate on the number of total relationships, which must be fulfilled, especially for the integrity constraint defined with this combination of mathematical symbols "==", the greater time will be required for data validation process. Another special case that needs to set a record straight is a *delete* operation. It may happen a special incident where we have a situation with the cardinality integrity constraint enabled with a *validate* keyword and with a condition where are used these mathematical symbols "==". If this situation occur, the integrity constraint will not allow to proceed the *delete* operation. This strict rule can be bypassed by disabling the integrity constraint, or enabling it with a *novalidate* keyword.

Endpoint requirements

The *Endpoint requirements* integrity constraint is designed only for relationships, not nodes. This integrity constraint should be served for defining certain requirements for relationships with specific limits. The *Endpoint requirements* is primarily determined for nodes, but the integrity constraint rule is defined on relationships and their *TYPE*. The integrity constraint should be able to define a relationship rule which requires a particular node with a certain label to act as a start node. This rule should be applicable for more nodes too where these nodes must also behave as start nodes. On the other hand we can turn the whole idea itself and instead of considering start nodes we can consider working with the end nodes. The next what is intended to use are hard-defined patterns in a similar kind of form *(x:NodeX)-[t:TYPE]->(y:NodeY)*, or the possibility that some relationship must not end in a certain node. A summary of those *Endpoint requirements* integrity constraint is shown in listing 2.19 with use of a pseudo Cypher syntax where the examples are inspired from this part of research 2.2.

Listing 2.19: Endpoint requirements - pseudo Cypher syntax

```
(1) CREATE CONSTRAINT ON (p:Person, o:Organisation)-
                    [o:OWNS]->() ASSERT exist(p, o)

(2) CREATE CONSTRAINT ON ()-[o:OWNS]->
    (v:Vehicle, b:Building, i:Item) ASSERT
                                  exist(v, b, i)

(3) CREATE CONSTRAINT ON (p:Person)-[o:OWNS]->
    (o:Organisation) ASSERT exist(p; o)

(4) CREATE CONSTRAINT ON (e:Employee)-[w:WORKS_FOR]->
    (e:Employee, m:Manager) ASSERT exist(e; e, m)

(5) CREATE CONSTRAINT ON ()-[o:OWNS]->(p:Person)
                                    ASSERT exist(!p)
```

The label separation in the *exist* clause is in general applicable for all examples listed in 2.19 and shown in figure 2.3.

If we want to define multiple labels at one time and in one place there is a solution in pseudo Cypher with comma separation. This applies for both the left and right node. For a relationship separation of those two different nodes is used a semicolon. The final formula in *exist* clause should look like in the mentioned example of the figure 2.3. Consider the first example (line number **(1)**) where is defined the integrity constraint which must check a rule for all nodes with a particular label *Person*, or *Organization* label. The integrity

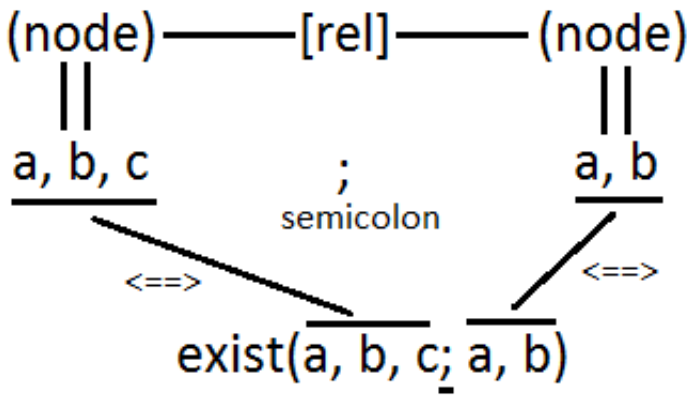


Figure 2.3: Endpoint requirement pseudo Cypher symbols legend.

constraint rule says that these nodes must have an outgoing relationship with type *OWNS*, the nodes must behave as the starting nodes, respectively. The second example **(2)** describes that incoming relationships with type *OWNS*, must end at nodes with labels *Vehicle, Building* and *Item*. The next **(3)** example strictly defines a border of the pattern rule which must be validated in the graph database. On the one side is a node with a label *Person* and outgoing relationship with type *OWNS* directed to the node with *Organization* label on the other side. Another example illustrates dependencies between nodes with labels *Employee* and *Manager*, and the last example prohibits the incoming relationship to the node with the *Person* label specification.

Regarding the problem of loading large amount of data was plentifully discussed in all preceding integrity constraint revisions. Here is also needed to have an ability on enabling and disabling this integrity constraint due to the performance. Then would follow these known steps *disable integrity constraint → load or update data → enable integrity constraint with validate or novalidate keyword*.

A deferred validation for this integrity constraint is really necessary. Because we need to count with a possibility of manual entry data where the nodes are firstly created and then joined with the relationships. If the integrity constrain was not set to a *deferred* mode, the validation process would immediately stopped with an integrity constraint violation message. Thus, the *insert* and *update* operations depend heavily on the integrity constraint execution time.

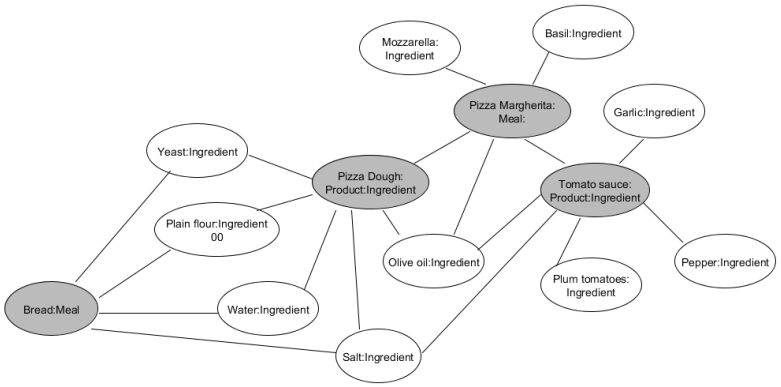


Figure 2.4: Label coexistence – the Ingredient example.

Label coexistence

The last integrity constraint is *Label coexistence*. In Neo4j graph database is possible to put zero, one, or more labels to nodes. This integrity constraint is concerned only for nodes and lays down limitations that some certain labels cannot be together in a single node, or vice versa. Then the node is, or is not allowed to be created. In research section 2.2 are provided two examples of the rules. The pseudo Cypher syntax can be seen is listing 2.20.

The explanation of the first example **(1)** is that a specific node cannot be simultaneously labeled with both the *Person* and *Organization* labels. By this integrity constraint restriction there only exist one possible way and it is either to create the node with the label *Person* or the node with the label *Organization*. The second example defines the rule that a node with a label *User* can only exist in a combination if a node with a label *Person* exist too and thus the node must contain the label *Person* and then it can contain the label *User*. For a better understanding of this behavior we have shown this in figure 2.4. The figure illustrates the necessary ingredients that are needed to achieve a certain product. We have shown the ingredients for making a Pizza Margherita or Bread, etc. In this example is used the *Label coexistence* integrity constraint where for a node with a label *Product* must simultaneously exist a label *Ingredient* which is exactly fitted to the second example mentioned in listing 2.20 for nodes with labels *User* and *Person*.

Listing 2.20: Label coexistence - pseudo Cypher syntax

```
(1) CREATE CONSTRAINT ON (p:Person, o:Organisation)
                                ASSERT exist (p || o)
(2) CREATE CONSTRAINT ON (p:Person, u:User) ASSERT
                                          exist (u && p)
```

As in all previous cases of integrity constraints where we were interested

in complexity during loading the large amount of data, here is also the right place to use this functionality of disabling and enabling integrity constraints in a chosen mode *enable validate*, or *enable novalidate*. In the worst case would have to be validated the whole database because we do not know whether the newly inserted, or updated data affected the database. If we take a look at an *insert* operation the complexity is in a constant time due to no relationships connections. The same applies for an *update* operation. If it is needed to change a label for the node, where one label must exist to another, the integrity constraint should violate the changes and will not allow it. For example, if we decide to change the *Ingredient* label with a *Foodstuff* label in figure 2.4. Our integrity constraint should evaluate this change as violating and would not allow the edit node labels. The *delete* operation is trivial with $O(1)$ complexity and there is no need to check anything.

Summary

This section dealing with the revision of proposed integrity constraints discussed in the research section 2.2 we mentioned some aspects which are needed to be thought over it if we want to put the integrity constraints into practice. We were very helped by our research about relational databases in section 2.1 where we found out such interesting issues which are really useful in enriching the basics of the integrity constraint definitions. Without these mentioned things which are applied for a long time in the relational databases, the work with integrity constraints in Neo4j graph database would be unthinkable.

We dealt with a wide range of aspects which are described in figure 2.2. Among these aspects belong a current, or pseudo Cypher syntax which covers actual, or possible integrity constraint definitions. During the discussion, which is pervaded through separated topics, we dealt with *usefulness*, and also *complexity* and its behavior to the graph database state. In a complexity part we focused on conducting known operations like *insert*, *update* and *delete* and how the graph database may behave depending on its capacity. In the end we took into account *time execution* and discussed use cases of using the *deferred*, or *immediate* modes.

Table 2.4 illustrates an asymptotic complexity for all discussed integrity constraint use cases. The table contains cases for both the *enable validate* and *novalidate* and either empty database or full database. When the database is loaded, or updated with data then must be validated under the certain integrity constraint. For that purpose the complexity is $O(n)$ where n is a total number of certain nodes that must be validated. But, there is one case represented by *Endpoint requirements* integrity constraint where is needed to validate by the total number of certain relationships where complexity is represented as $O(|E|)$, letter E like edges. All the other operations are almost $O(1)$, but we must be careful of hidden influencers. The reason why is as follows. For ex-

ample we can define an integrity constraint for *Endpoint requirements* where a pattern would look like *(:Employee)-[:WORKS_FOR]->(:Employee:Manager)*. If we decided to rename label ":Employee:Manager" to "Employee", the integrity constraint would not allow this operation because it might be needed to validate all relationships with type *WORKS_FOR* and it is not complexity $O(1)$ as we mentioned in the table 2.4, but $O(|E|)$. It is needed to realize that only one *update* operation would start a time-consuming validation process which is really not $O(1)$. These dependencies we can find at integrity constraints like *Cardinality requirements* or *Required relationships*. Among "so-called" safe integrity constraints we can only include *Mandatory property constraints*, *Property value limitations* and *Label coexistence* where the complexity should be the same as is illustrated in the table 2.4 for all possible operations without any hidden influencers.

The last thing what is absolutely missing in the integrity constraint proposals is lack of an integrity constraint management which would be nice to have for getting information about activated integrity constraints in the graph database. This issue is discussed in the next design chapter 3.

2.4 Requirements and assignment

Our assignment is to propose a new Cypher Query Language syntax which will support a selected subset of integrity constraints. An implementation of the new designed Cypher Query Language will be in Java. The prototype implementation with exposed API interface will extend Neo4j graph database engine about the subset of new integrity constraints.

Functional requirements

- Design new Cypher Query Language syntax for defining the selected integrity constraints.
- The prototype implementation will cover three types of integrity constraints orientated only for a node validation such as:
  - **Node property uniqueness** validation behavior for a single property value and multiple property values (exactly for a combination of a pair properties)
  - **Mandatory properties** to provide properties that must not be NOT NULL
  - **Property value limitations** to restrict particular properties by some limits with help of the regular expression, datatype restriction, etc.
- Integrity constraints will be able to define both manually in code and in file where is a possibility to load defined integrity constraints that are stored in a JSON structure.

Table 2.4: Complexity of integrity constraints validation

Loading large amounts of data								
	Enable validate		Enable novalidate					
Constraint name/Action	Empty DB	Full DB	Empty DB	Full DB	Insert	Update	Delete
Node property uniqueness	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$
Mandatory property constraints	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Property value limitations	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Required relationships	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Cardinality requirements	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Endpoint requirements	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$	$O(1)$	$O(1)$	$O(1)$
Label coexistence	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$

- The provided prototype implementation API will provide database Schema definition in the form of integrity constraints.
- The database schema will be maintained in a hierarchical structure divided by nodes and relationships.
- The prototype implementation will offer integrity constraints management system; already returned defined integrity constraints will be represented in a JSON format.
- The implementation will not support an integrity constraint definition in Cypher and communication with a database via REST API.
- The implementation itself will not support parsing of defined integrity constraint in Cypher.
- The prototype implementation after having done integrity constraint definition will not provide simultaneously creation of such indexes for speeding-up of searching in a graph database.

Non-functional requirements

- The prototype of a schema definition by integrity constraints will be implemented in Java.
- The prototype will be implemented as an API for an embedded mode of a Neo4j graph database engine.
- The prototype will use GraphAware framework for testing.

Limitations

The provided implementation will act is a prototype dealing with a possible way which integrity constraints may proceed. We cannot rely on fully functional life-cycle of processing the integrity constraints. Especially, we excluded a definition of an integrity constraint through a Neo4j web interface because that functionality is out of scope of this diploma thesis. We also will not be bother entering and parsing integrity constraint queries entered in Cypher language.

Chapter 3

Cypher syntax design

This chapter serves for introducing an extension of the *Cypher Query Language*. We use all knowledge we earned in previous chapters. With regard to the integrity constraints itself and their fundamental functionality, we are mostly inspired from the relational databases where this all kind of stuff works for a long time. In this chapter we design one of the possible ways how to define a new integrity constraint rule in Cypher. The design is based on a syntax which provided us the whole integrity constraints revision section (2.3). However, at first, we need to introduce common definitions used in databases and also we need to find out the new formal definitions for our purpose.

3.1 Comprehensive definitions

Schema (Relational databases) A database schema is a skeleton structure that represents a logical view of an entire database. The schema defines how the data, especially in relational databases, will be organized and how the relations between the data are organized. Thus, database schema describes entities and relationships among them and formulates all integrity constraints that we want to have applied on the data [47].

Figure 3.1 shows the one possible basic example of the database schema where is a description of a *room reservation* for the relational databases. However, graph databases are fundamentally schema-free which means that they do not have any similar strict schema description like relational databases.

Schema (Neo4j graph database) Is a persistent database state that describes available indexes and enabled integrity constraints for the property graph [49].

We defined what the schema stands for the graph database Neo4j. The persistent database state means that the database preserves previous amount of data for further processing, holds indexes for better querying and time savings, and enables integrity constraints for allowing restrictions. The entire

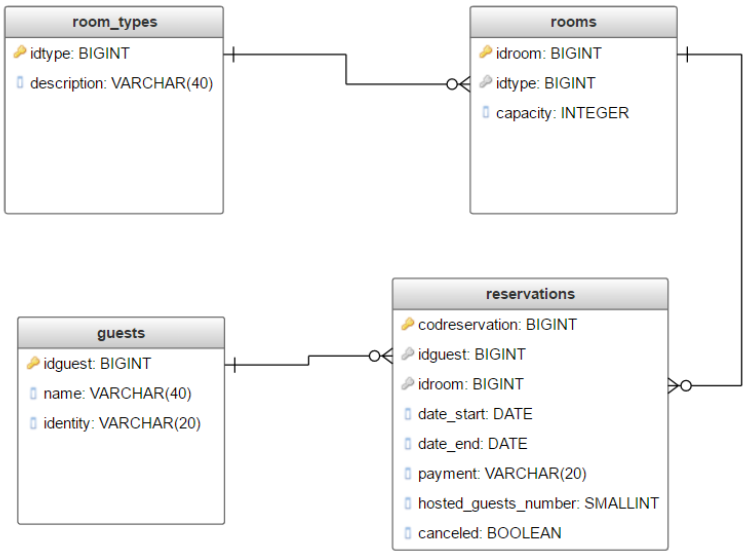


Figure 3.1: Example of schema in a relational database [48].

thesis deals with the integrity constraints. In view of the fact that Neo4j is *schema-free* and customers in a certain areas become accustomed to the benefits which the relational databases have. They wanted to make certain parts of Neo4j to be limited in a way of restricting schema freedom in a form of integrity constraints wherefrom come the name **Schema Enforcement**.

Our proposal research conducted at 2.2 tells us that we must deal with nodes and relationships in the property graph because for any integrity constraint implies that needs to have somewhere defined the integrity constraint rule. Therefore, can be useful to have a top schema organization structure for integrity constraints. Such needed structure we can see in figure 3.2.

Schema Configuration hierarchical structure is a structure which holds a configuration for integrity constraints separated into both the *Nodes Configuration* and *Relationship Configuration*. Each separated part has its own default template for segregation constraint rules.

In simple terms a *Schema Configuration hierarchical structure* consists of top level component called **Schema Configuration**. Schema Configuration has such a role in taking care of **Nodes** and **Relationships Configurations**. These determined configurations are needed to keep integrity constraint rules apart because in research section 2.2 we mentioned that exist integrity constraints which are only determined to either nodes or relationships and this option should be very suitable for us. Each configuration part (the Node and Relationship) has intended to have a default pre-defined template from which an inheritance (dashed rectangles in figure 3.2) can be used.

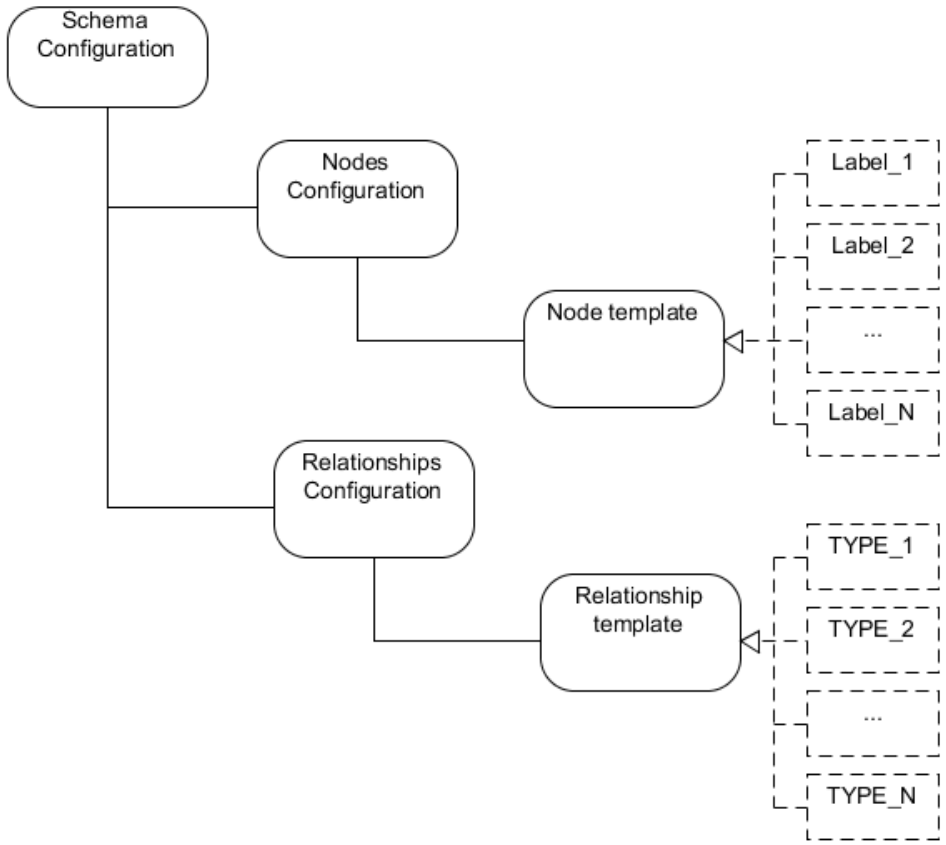


Figure 3.2: Schema Configuration hierarchical structure.

Node and Relationship template is a fundamental pre-defined class structure which has defined all the things that are needed to be applicable across all nodes and relationships in a graph database, respectively.

How could, such a template should look like under this definition? If we take a look at figure 3.3 we can find out very similar things that are among the *Node* and *Relationship* templates. This concept is divided into three parts and it is possible that through the design section will be changed.

First part is about a naming convention. Neo4j supports that for the nodes is not required to give them a name (i.e. a node label) and also it is possible that the node can have more than one label. So, to be able to distinguish the names under the newly defined integrity constraints, there is a visible sign (0..*) for our information that it is possible to assign zero, or more node labels. For a relationship template is a situation a little bit different where is possible only one action and it is to assign a relationship with one TYPE, this is symbolized by a star sign (*) as a necessity.

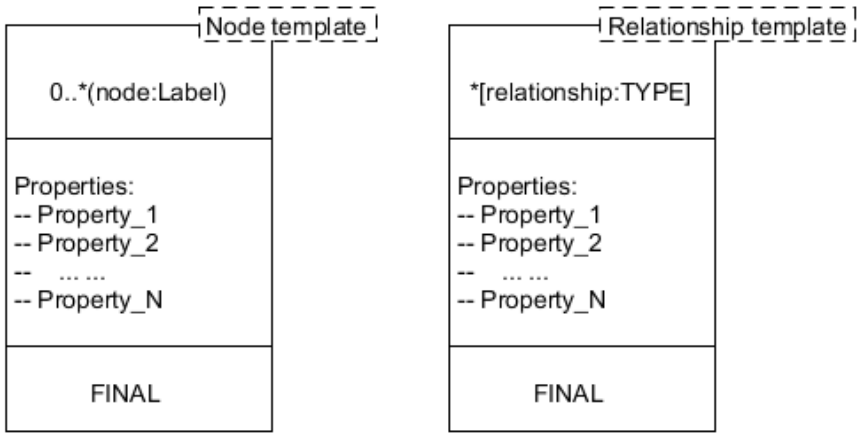


Figure 3.3: Node and Relationship template in detail.

The second part is about properties. This part is the same for both the Node and Relationship templates. There are stored property records from the integrity constraint rule.

The last one is a FINAL part which has a special functionality. The FINAL is applicable for both templates. If the FINAL is set to ON, it will not be possible to make any changes such as property addition, deletion, or rename. For the naming convention is not possible to rename the node label or the relationship TYPE. Afterwards, we are able to use an inheritance to create the precise records (i.e. dashed rectangles in figure 3.2) for the integrity constraint rule from the pre-defined template.

3.2 Cypher Query Language syntax for integrity constraints

The section contains discussion and designs the Cypher Query Language syntax for those integrity constraints proposed in chapter 2 section 2.2. The draft of the appropriate CQL we take into account all our knowledge that we have already acquired in the previous chapters and sections concerning the research, integrity constraint proposals, and also our revision. This section is split into parts according to those integrity constraints, we present a brief introduction of the problem and then we design a new extension of how the integrity constraint problem could be solved in Cypher Query Language.

Before we begin with the new design of Cypher Query Language, firstly, we introduce a skeleton of Cypher under which other kind of integrity constraints are derived. Our objective was to keep the existing Cypher Query

Language syntax, so that the new Cypher syntax were not so much different from the old one and also was immediately possible easy to use it and understand. The general skeleton of Cypher for all necessary clause operations such as CREATE, MATCH, EDIT, REMOVE, DROP and DISABLE are illustrated in listing 3.1.

Listing 3.1: Integrity constraint skeleton - Cypher

```
(1) CREATE CONSTRAINT (name:'ic_name') ON
        (PATTERN) ASSERT ACTION(properties)
         OPTIONS(enable:'{VALIDATE|NOVALIDATE}',
                 validation:'{IMMEDIATE|DEFERRED}',
                 delete: '{RESTRICT|CASCADE}',
                 update: '{RESTRICT|CASCADE}',
                 final: '{FALSE|TRUE}');

(2) MATCH (all_constraints) WHERE name = 'ic_name'
    SET (PATTERN) ASSERT ACTION(properties)
    OPTIONS(enable:'{VALIDATE|NOVALIDATE}',
                 validation:'{IMMEDIATE|DEFERRED}',
                 delete: '{RESTRICT|CASCADE}',
                 update: '{RESTRICT|CASCADE}',
                 final: '{FALSE|TRUE}');

(3) DROP (all_constraints) WHERE name = 'ic_name';

(4) DISABLE (all_constraints) WHERE name = 'ic_name';

(5) ENABLE (all_constraints) WHERE name = 'ic_name';
```

Line identified by the numeral **(1)** as a part of the skeleton is a creation of a new integrity constraint in Cypher. It is composed of several components:

- **The integrity constraint name** as a required unique property.

  The current integrity constraint Cypher syntax in Neo4j is based on the fact that there is not any register place for already defined integrity constraints and their validation policy. This situation facilitates the fact that integrity constraints in Neo4j are not much widespread at the form we would like to have. If we return to our integrity constraint situation, the design expanded in new functionalities without any simple integrity constraint identification would have difficulty in using them. Therefore, we must distinguish the integrity constraints by name for the next Cypher clauses we want to use.

- **PATTERN** part as a required property.

This section is reserved for determining the pattern under which integrity constraint validation should be performed. Typical patterns are: *(n:NodeLabel)* for nodes, *[t:TYPE]* for relationships or a combination of both, for example, *()-[t:Type]->(n:NodeLabel)*. Where the last case deals with this situation. From all the arbitrary nodes from which goes a type relationship [:TYPE] must only go to the such nodes which are labeled as (:Node).

- **ACTION(properties)** part as a required property.

  The *ACTION* clause serves for defining nodes, or relationship properties like *email, username, etc.*, that should be validated with a condition for some integrity constraint cases. The ACTION itself can acquire **UNIQUE**, or **EXISTS** values.

- **OPTIONS(options)** the last part is optional.

  An assigning pair style (or key-value) is inspired from basic Cypher syntax where for a particular property key is assigned a particular value. Due to our research in chapter 2 we collected for our purpose five properties that should be placed to the OPTIONS part. Between those keys belong *enable, validation, delete, update* and *final*:

  - enable: '{VALIDATE|NOVALIDATE}'

    Enable with **VALIDATE** starts immediately validating the whole database. Enable integrity constraint with **NOVALIDATE** action is applicable only for newly inserted data, or newly updated data, present data will not be validated.

  - validation: '{IMMEDIATE|DEFERRED}'

    This refers to a validation time. If we want to postpone the integrity constraint validation, we will pick the **IMMEDIATE** keyword, otherwise the **DEFERRED** keyword.

  - delete/update: '{RESTRICT|CASCADE}'

    This **RESTRICT** action prevents from the delete, or update operation in a referenced node. The **CASCADE** action for the delete, or update means that if a property in a a certain node is deleted, then the records in the child nodes will be deleted automatically too.

  - final: '{FALSE|TRUE}'

    If we do not wish in order to have allowed adding additional properties to the nodes and relationships, we will assign a **TRUE** sign to the final property. Otherwise if we want to, we will let the default value which is a **FALSE**.

    To illustrate the main idea lets show it in an example for a certain node labeled with User. Consider the node with the **User** label

and following properties *username*, *password* and *email*. We want to make it impossible enriching the node with label User in the future with additional properties. Thus, we will create the integrity constraint with **Final:'TRUE'**. Since this time it will not be able to change the node definition given by the integrity constraint for the node label User. If we would like to enrich the node with a property *phoneNumber*, the integrity constraint will not allow it and firstly we will have to disable, or drop the integrity constraint. The same situation applies if we wanted to remove some property from the User node, e.g. *password*, the integrity constraint would not allowed it.

The OPTIONS part is an optional and there is not needed to fill any key with a particular value. In curly braces are placed, for each key, two values with a sign pipe "|" separation. For the keys the first value in curly braces is used as a default value. For *delete* and *update* keys with values *restrict* and *cascade* are used only for those integrity constraint which has pattern syntax defined in this and similar forms *()-[t:Type] ->(n:NodeLabel)-[t:Type]->(n:NodeLabl)*.

Line (2) in listing 3.1 demonstrate a skeleton for an integrity constraint edition where is used a standard Cypher syntax with improvements mentioned above, i.e. the line (1). What is really important and is noteworthy is an integrity constraint management itself. Neo4j does not support any integrity constraint management. Our approach is simple in using the standard Cypher clause MATCH where we query a set where the integrity constraints are stored.

The next proposed syntax DROP on line (3) is for removal the whole integrity constraint rule which disables for good the validation policy of that certain integrity constraint rule. Line (4) is designed for disabling integrity constraint for a certain period of time then it is needed, with line (5), to enable the integrity constraint again with help of the ENABLE clause, which in a default mode enables the integrity constraint as **VALIDATE** which means that will be validated only new incoming data by the integrity constraint, not the data which were already loaded in the database.

It might seem that the command design for DROP, DISABLE and ENABLE clauses could be done via MATCH clause. Yes, it is true, but the reason why it is splitted is more than obvious. We were trying to avoid at probably the most frequently used commands using spreading Cypher syntax. Where the use would be higly impractical and would complain the work. Once again, for a better integrity constraint recognition is used the integrity constraint naming convention.

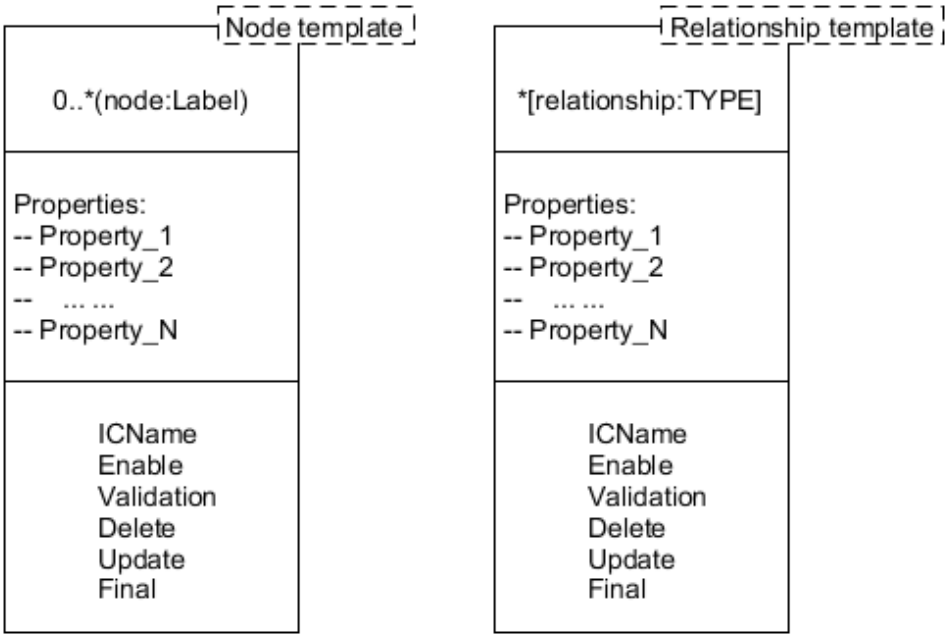


Figure 3.4: Node and Relationship template revised in detail.

Node and Relationship template revised

In definition section 3.2 we introduced a basic structure of Node and Relationship templates which are parts of a *Schema configuration*. The structure in those templates was designed before designing the new Cypher Query Language syntax and we discovered that it is not sufficient enough. Therefore, we made another new template proposal which covers all the required information for storing the integrity constraints rules. Figure 3.4 shows an updated version of Node and Relationship templates where is considered an integrity constraint *name*, and other matters from OPTIONS part in Cypher design which are the *Enable, Validation, Delete, Update* and *Final* properties.

Node property uniqueness

How it was said in the constraint revision section 2.3, the *Node property uniquenes* integrity constraint exists in its simplified way. This integrity constraint solves uniqueness of the nodes properties. Firstly fulfill a unique value for separated properties, like the email or username property validation for a User node or checking a combination of unique values where as an example was mentioned a label Person with properties firstName and lastName and where must not appear in a graph database the person with the same credentials twice.

Listing 3.2: Node property uniqueness - Cypher proposal

```
(1) CREATE CONSTRAINT (name:'unqMail') ON
        (u:User) ASSERT UNIQUE(u.email);

(2) CREATE CONSTRAINT (name:'unqMail') ON
        (u:User) ASSERT UNIQUE(u.email)
         OPTIONS(enable:'VALIDATE',
                 validation:'DEFERRED',
                 final: 'TRUE');

(3a) CREATE CONSTRAINT (name:'unqComb') ON
        (p:Person) ASSERT UNIQUE(p.firstName &&
                                 p.lastName);

(3b) CREATE CONSTRAINT (name:'unqComb') ON
        (p:Person) ASSERT UNIQUE(p.firstName,
                                 p.lastName);

(4) MATCH (all_constraints) WHERE name = 'unqMail'
    SET (p:Person) ASSERT UNIQUE(p.email)
    OPTIONS(enable:'{VALIDATE|NOVALIDATE}',
                 validation:'{IMMEDIATE|DEFERRED}',
                 delete: '{RESTRICT|CASCADE}',
                 update: '{RESTRICT|CASCADE}',
                 final: '{FALSE|TRUE}');

(5) DROP (all_constraints) WHERE name = 'unqMail';

(6) DISABLE (all_constraints) WHERE name = 'unqMail';

(7a) MATCH (all_constraints) WHERE name = 'unqMail'
     SET OPTIONS(enable:'VALIDATE');

(7b) ENABLE (all_constraints) WHERE name = 'unqMail';
```

In listing 3.2 is shown several examples how to create an integrity constraint rule under a specific assigned constraint name. There are creational cases with or without OPTIONS part (line numbers 1 to 3b). Lines (3a) and (3b) shows two different approaches of unique property combination. The first one uses && signs and the second one uses a comma. Next following examples cover edit with help MATCH and SET commands, then DROP, DISABLE and enable cases. The enable cases have two approaches. The first consist of the MATCH and SET clauses combination, the second demonstrates enable **VALIDATE** only with the use of the ENABLE clause.

The next any subsequent examples defining the integrity constraints with the new Cypher syntax is not covered with the OPTIONS part (except for some cases) inasmuch as would be repeated the same Cypher syntax. An explanation of what each part of the OPTIONS part means can be found in section 3.1 under the Integrity constraint skeleton. Also, all integrity constraint examples are taken from research section 2.2 of the analysis chapter to demonstrate them in the Cypher language.

Mandatory properties

This kind of integrity constraint exists too. It is allowed to use it only if we have Neo4j Enterprise edition, not the Community version. The integrity constraint carries feature known from the relational databases as NOT NULL. The Mandatory property integrity constraint should be specified for both nodes and relationships properties.

Listing 3.3: Mandatory properties - Cypher proposal

```
(1) CREATE CONSTRAINT (name:'notNullMail') ON
        (u:User) ASSERT EXISTS(u.email);

(2) CREATE CONSTRAINT (name:'notNullMail') ON
        (u:User) ASSERT EXISTS(u.email)
         OPTIONS(enable:'VALIDATE',
                 validation:'DEFERRED');

(3) CREATE CONSTRAINT (name:'notNullDist') ON
        [r:ROAD] ASSERT EXISTS(r.distance);

(4) CREATE CONSTRAINT (name:'notNullDist') ON
        [r:ROAD] ASSERT EXISTS(r.distance);
         OPTIONS(enable:'VALIDATE',
                 validation:'DEFERRED');

(5) MATCH (all_constraints) WHERE name = 'notNullMail'
    SET (p:Person) ASSERT EXISTS(p.email)
    OPTIONS(enable:'{VALIDATE|NOVALIDATE}',
                 validation:'{IMMEDIATE|DEFERRED}',
                 delete: '{RESTRICT|CASCADE}',
                 update: '{RESTRICT|CASCADE}',
                 final: '{FALSE|TRUE}');

(6) DROP (all_constraints) WHERE name = 'notNullMail';
```

```
(7) DISABLE (all_constraints) WHERE name =
    'notNullMail';

(8a) MATCH (all_constraints) WHERE name = 'notNullMail'
     SET OPTIONS(enable:'VALIDATE');

(8b) ENABLE (all_constraints) WHERE name =
     'notNullMail';
```

Listing 3.3 also follows the RFP paper with integrity constraint proposals and demonstrates the functionality of defining the integrity constraint rule through Cypher language. The integrity constraint can be created for both nodes and relationships. The other Cypher examples *MATCH, DROP, DISABLE* and *ENABLE* are shown only for the first case (line number **(1)**) because the next examples would repeat.

Property value limitations

The property value limitations integrity constraint is responsible for a proper validation whether the properties meet the requirements defined in the Cypher language. For example a certain property must be of a Boolean datatype or must not be negative, etc.

Listing 3.4: Property value limitations - Cypher proposal

```
(1) CREATE CONSTRAINT (name:'positiveDist') ON
        [r:ROAD] ASSERT EXISTS(r.distance > 0);

(2) CREATE CONSTRAINT (name:'regexMail') ON
        (u:User) ASSERT EXISTS(u.email AS
                        "[a-z]?@[a-z].+[a-z]");

(3) CREATE CONSTRAINT (name:'UABool') ON
        (u:User) ASSERT EXISTS(u.active AS BOOLEAN);

(4) CREATE CONSTRAINT (name:'locPos') ON
        (u:User) ASSERT EXISTS(u.active AS LIST<POINT>);

(5) MATCH (all_constraints) WHERE name = 'positiveDist'
    SET [r:ROAD] ASSERT EXISTS(r.distance >= 0);

(6) DROP (all_constraints) WHERE name = 'positiveDist';

(7) DISABLE (all_constraints) WHERE name =
    'positiveDist';
```

```
(8) MATCH (all_constraints) WHERE name = 'positiveDist'
    SET OPTIONS(enable:'NOVALIDATE');
```

The integrity constraint definition on line (3) where is defined a node with a label user must exist a property *active* which must be of the Boolean datatype. The next property values are those values that describes a Java Language Specification.

Required relationships

The Required relationships integrity constraint defines for nodes with a particular label that must be linked with a particular relationship **TYPE**.

Listing 3.5: Required relationships - Cypher proposal

```
(1) CREATE CONSTRAINT (name:'outRRPerson') ON
        (p:Person) ASSERT EXISTS(p-[:LIVES_AT]->());

(2) CREATE CONSTRAINT (name:'inRRFlow') ON
        (s:Sink) ASSERT EXISTS(s<-[:FLOW]-());

(3) CREATE CONSTRAINT (name:'RRNeighbl') ON
        (p:Place) ASSERT EXISTS(p-[:NEIGHBOUR]-());

(4) MATCH (all_constraints) WHERE name = 'outRRPerson'
    SET (p:Person) ASSERT EXISTS(p-[:WORK_AT]->());

(5) DROP (all_constraints) WHERE name = 'outRRPerson';

(6) DISABLE (all_constraints) WHERE name =
    'outRRPerson';

(7) MATCH (all_constraints) WHERE name = 'outRRPerson'
    SET OPTIONS(enable:'NOVALIDATE');
```

Cardinality requirements

The Cardinality requirements integrity constraint is almost the same as the *Required relationships* integrity constraint. This one requires the cardinality, thus the minimum and maximum number of relationships with a certain **TYPE** for a given node with a certain label.

Listing 3.6: Cardinality requirements - Cypher proposal

```
(1) CREATE CONSTRAINT (name:'cardRCitizen') ON
    (s:SwedishCitizen) ASSERT EXISTS
                              (s-[MARRIED_TO]-() <= 1);

(2) CREATE CONSTRAINT (name:'cardRparent') ON
    (p:Person) ASSERT EXISTS(p<-[PARENT_OF]-() == 2);

(3) CREATE CONSTRAINT (name:'buddMonk') ON
    (b:BuddhistMonk) ASSERT EXISTS
    (b-[:OWNS]->() <= 14);

(4) MATCH (all_constraints) WHERE name = 'cardRCitizen'
    SET (s:CzechCitizen) ASSERT
    EXISTS(p-[:WORK_AT]->());

(5) DROP (all_constraints) WHERE name = 'cardRCitizen';

(6) DISABLE (all_constraints) WHERE name =
    'cardRCitizen';

(7a) MATCH (all_constraints) WHERE name = 'cardRCitizen'
     SET OPTIONS(enable:'VALIDATE');

(7b) ENABLE (all_constraints) WHERE name =
     'cardRCitizen';
```

Endpoint requirements

The Endpoint requirements integrity constraint is defined in Cypher language on relationships. This integrity constraint defines the specific rules for relationships where must or must not start or end in the certain nodes.

Listing 3.7: Endpoint requirements - Cypher proposal

```
(1) CREATE CONSTRAINT (name:'erOwnsOutgoing') ON
    (p)-[:OWNS]->() ASSERT
    EXISTS(p:Person|Organisation);

(2) CREATE CONSTRAINT (name:'erOwnsIncoming') ON
    ()-[:OWNS]->(v) ASSERT
    EXISTS(v:Vehicle|Building|Item|Organisation);
```

```
(3) CREATE CONSTRAINT (name:'erOwnsStrict') ON
    (p)-[:OWNS]->(o) ASSERT
    EXISTS(p:Person;o:Organisation);

(4) CREATE CONSTRAINT (name:'erWorkStrict') ON
    (e)-[:WORKS_FOR]->(m) ASSERT
    EXISTS(e:Employee;m:Employee:Manager);

(5a) CREATE CONSTRAINT (name:'erOwnsNegative') ON
     ()-[:OWNS]->(p) ASSERT EXISTS(!p:Person);

(5b) CREATE CONSTRAINT (name:'erOwnsNegative') ON
     ()-[:OWNS]->(p) ASSERT NOT EXISTS(p:Person);

(6) MATCH (all_constraints) WHERE name =
    'erOwnsOutgoing' SET (p:Person) ASSERT
    EXISTS(p-[:LIVE_AT]->());

(7) DROP (all_constraints) WHERE name =
    'erOwnsOutgoing';

(8) DISABLE (all_constraints) WHERE name =
    'erOwnsOutgoing';

(9) MATCH (all_constraints) WHERE name =
    'erOwnsOutgoing' SET OPTIONS(enable:'NOVALIDATE');
```

Label coexistence

The last integrity constraint is Label coexistence. The integrity constraint specifies that nodes with a certain labels cannot exist, or can exist with some required condition.

Listing 3.8: Label coexistence - Cypher proposal

```
(1) CREATE CONSTRAINT (name:'labCoexist1') ON
    (p:Person:Organisation) ASSERT
    EXISTS(p:Person||p:Organisation);

(2) CREATE CONSTRAINT (name:'labCoexist2') ON
    (p:Person:User) ASSERT EXISTS(p:Person && p:User);
```

```
(3) MATCH (all_constraints) WHERE name = 'labCoexist1'
    SET (p:Person:Organisation) ASSERT EXISTS
    (p:Person && p:Organisation);

(4) DROP (all_constraints) WHERE name = 'labCoexist1';

(5) DISABLE (all_constraints) WHERE name =
    'labCoexist1';

(6a) MATCH (all_constraints) WHERE name = 'labCoexist1'
     SET OPTIONS(enable:'VALIDATE');

(6b) ENABLE (all_constraints) WHERE name =
     'labCoexist1';
```

3.3 Summary

This chapter led us through the whole problem of defining a new Cypher Query Language syntax. At the beginning we introduced the most essential definitions. We defined a difference between schemas in relational databases and a graph database, especially with focus to Neo4j. Then, we defined a hierarchical structure of our schema configuration for making records from defined integrity constraints with help of Cypher language. We mentioned that the hierarchical structure has templates for a node and relationship where are afterwards used to inheritance. Our next task was to propose a new syntax for integrity constraint definition in Cypher Query Language. The new syntax is basically formed as a skeleton where are considered all the possible use cases. After specification of the Cypher syntax we had to made a few changes in our theoretical part concerning the *Node and Relationship templates* 3.2. Then continued the application of the new Cypher syntax to the examples of integrity constraints defined in the RFP paper which were discussed in the research section 2.2.

Chapter 4

Realisation

At the end of the analysis chapter we delimited in the requirements section 2.4 what types of integrity constraints will be implemented in a prototype solution. The issue is about implementing the integrity constraints for validation of nodes and its surroundings. We speak about **Node property uniqueness, Mandatory properties** and **Property value limitations**. In view of the fact that we talk about the prototype implementation some designed functions will not be available. What is not aim of a development of the integrity constraints is an executing those defined integrity constraints in Cypher language because the processing, solving, parsing, capturing and filtering of the particular Cypher syntax, what kind of defined integrity constraints in Cypher would be routed further, and another kind routed to the pre-processing phase, is out of scope of this thesis. The whole implementation is based on using Neo4j graph database in an embedded mode. For defining integrity constraints and their management is used a developed *SchemaConfigurationAPI* which provides an interface to use convenient methods in cooperation with structures in code (or in .json file) for keeping defined integrity constraints. This chapter guides us through the implementation background and what core functions are used in the *SchemaConfigurationAPI* implementation itself with examples of use.

4.1 Implementation background

Neo4j Core API

In section about Neo4j graph database 1.2 we mentioned the high-level architecture parts of Neo4j graph database, and how nodes and relationships are physically stored. The selected integrity constraints are implemented for Neo4j graph database engine. Neo4j in large part is implemented in Java, therefore also Java was used for implementation of integrity constraints. The

most relevant part of the Neo4j architecture is *Neo4j Core API* which is an imperative Java API responsible for a communication with the graph database.

The most important part is a *GraphDatabaseService* interface which is used as a mediator between a code and running Neo4j database instance. This interface provides many methods such as querying and updating data in the graph database. There are also supported actions for the node and relationship creation, traversing a graph by nodes or relationships, transaction execution, and many more. The ACID properties to be fulfilled, the transactions have designed an interface for doing them in right way. Listing 4.1 shows the use of the transaction interface where all operations are placed in a try-catch block.

Listing 4.1: An example of a transaction in Java

```
GraphDatabaseService database;
...
try (Transaction tx = database.beginTx())
{
        // doing graph operations
        ...
        tx.success();
} catch (Exception ...)
  {
        // Handling errors
        ...
  }
```

The next two important interfaces which are provided by Neo4j Core API are the *Node* and *Relationship interfaces*. The Node interface supports many methods that cover all possible operations to be done on the nodes. These operations for a Node can be access to the node labels, its properties, and incoming or outgoing relationships. Operations for a Relationship are getting the nodes, type of a certain relationship, its properties, etc. Both Node and Relationship support retrieval operation (i.e. manual traversing graph by a programmer) which can be handled by an Iterator pattern.

The *GraphDatabaseService* interface also enables to execute queries using the Cypher language. For this possibility was designed an *execution* method. This method takes a Cypher command as a String parameter and returns as a result an instance of a Result class. This method does not need to be wrapped in the try-catch block because, by default, is executed within the transaction [3].

Neo4j Transaction Event API

The **Neo4j Transaction Event API** implements a *TransactionEventHandler* interface which must be registered within a database instance provided by the *GraphDatabaseService* interface. With Neo4j Transaction Event API we can use methods like *beforeCommit*, *afterCommit* and *afterRollback* which have provided an instance of TransactionData (figure 4.1). Listing 4.2 shows a skeleton in Java and how to use the TransactionEventHandler.

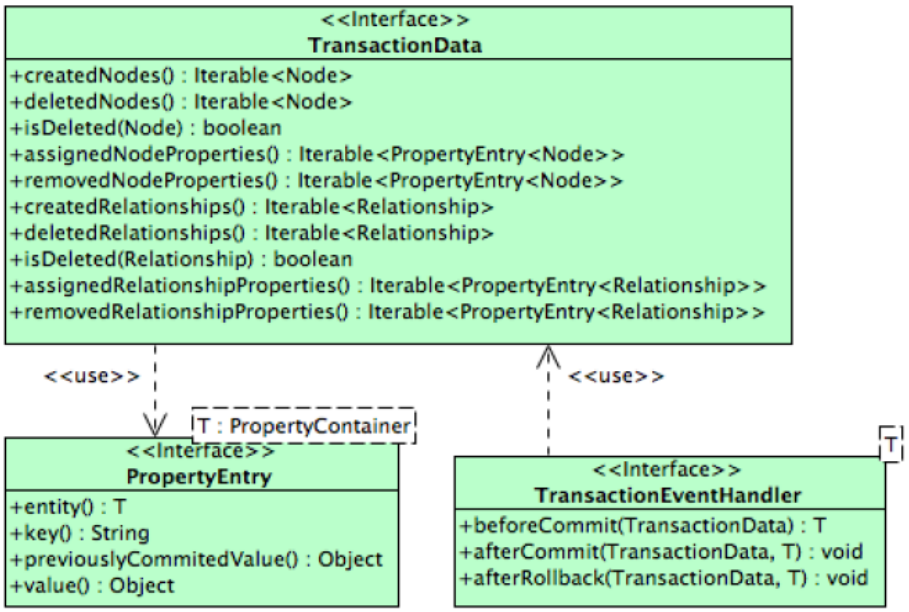


Figure 4.1: Neo4j Transaction Event Handler API [8].

Listing 4.2: An example of a transaction in Java

```
GraphDatabaseService database;
...
database.registerTransactionEventHandler(new
   TransactionEventHandler<Void>()
{
        @Override
        public Void beforeCommit(TransactionData
           transactionData) throws Exception
        {
                //Operations to be performed before
                   commit
                return;
        }
```

```
        @Override
        public void afterCommit(TransactionData
           transactionData, Void aVoid)
        {
                //Operations to be performed after
                   commit
                return;
        }
        @Override
        public void afterRollback(TransactionData
           transactionData, Void aVoid)
        {
                //Operations to be performed after
                   rollback
                return;
        }
});
```

The *beforeCommit* method serves us for making changes on our own. Due to the *transactionData* instance we are able to control and ensure which nodes and relationships have been created and deleted in the transaction before the whole transaction is committed. This method is widely used for ensuring and validating data under the defined integrity constraints rules, and where we are able to enforce a database schema to validate data, and then if data violate the database schema, we will return an appropriate error message. The other methods are self-explained where we can also perform some operations.

4.2 Implementation of integrity constraints

The implementation model described in figure 4.2 describes a way how the integrity constraints are managed and kept in a database schema. The implementation of the selected integrity constraints involves several classes. As we can see, we proceed from the bases of the proposal suggested in our design section 3.2. The core and most important class is a *SchemaConfigurationAPI* and its relations with another classes supported the whole functionality. The *SchemaConfigurationAPI* interface is designed for Neo4j graph database as a single instance. It was achieved by a Singleton design pattern where the instance of *SchemaConfigurationAPI* can be obtained by calling a public method *getInstance*. The *SchemaConfigurationAPI* handles all operations needed to start an integrity constraint validation. The class provides methods to manage defined integrity constraints and print them out, enforce data on the bases of integrity constrains, and register configurations for integrity constraints defined for nodes or relationships. For data enforcing is needed to provide the *enforce* method an instance of *TransactionData* and an instance of

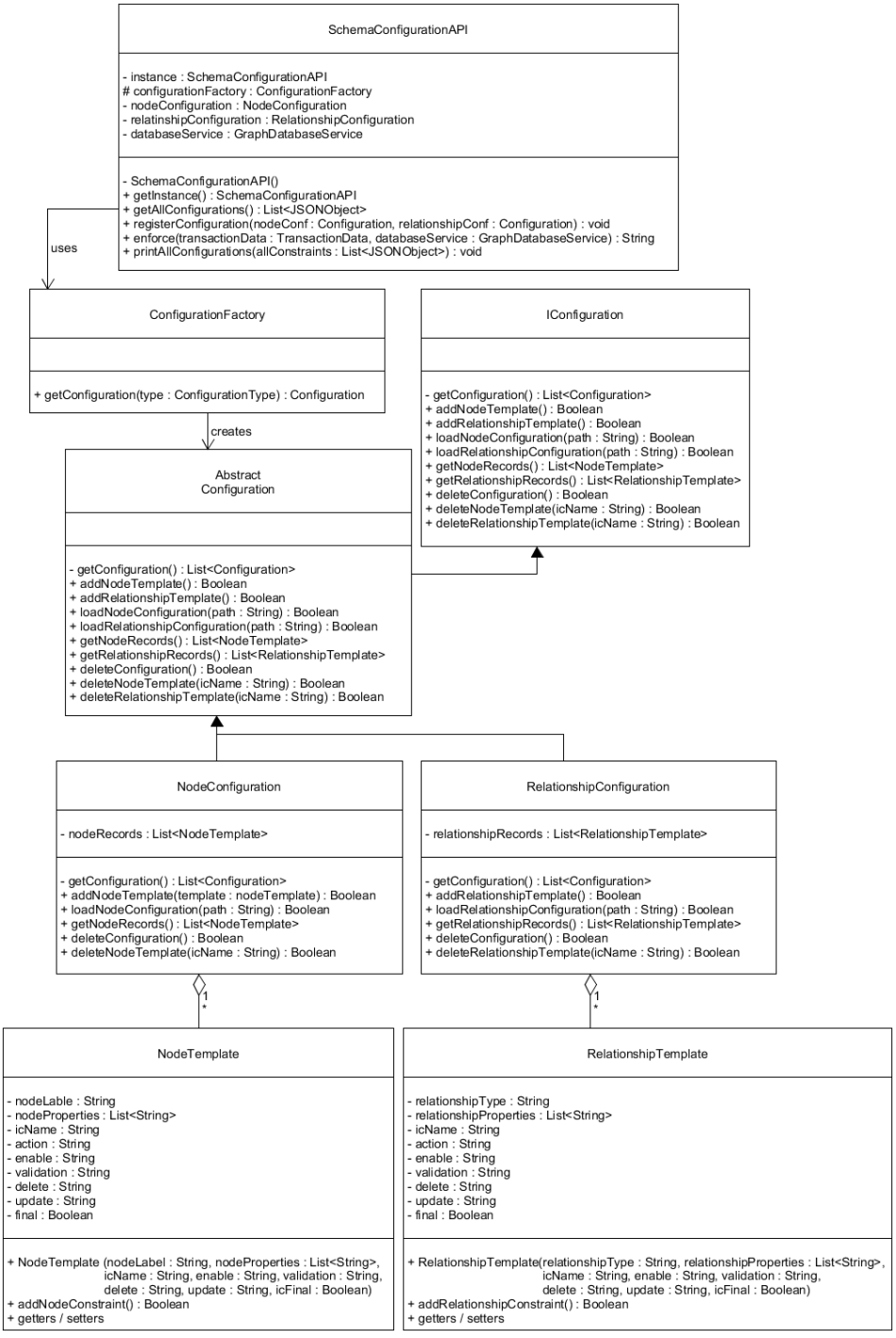


Figure 4.2: UML class diagram – Schema configuration API model.

Neo4j database a *GraphDatabaseService*. A schema configuration records must be registered by a *registerConfiguration* method where is needed to be passed *Configurations* for both nodes and relationships. If one of node or relationship configuration is not necessary, then just instead of them fill a null value.

It is allowed to create only a *NodeConfiguration* and *RelationshipConfiguration*, not the *Configuration* itself because it is implemented as an abstract class. Anyone, who want to define integrity constraints via the *NodeConfiguration* or *RelationshipConfiguration* is applied a Factory design pattern for a better selection of a required configuration type. Thus the *NodeConfiguration* and *RelationshipConfiguration* inherit application logic from the *Configuration* class. The *NodeConfiguration* holds every node record for each defined integrity constraint kept in a *NodeTemplate*. The same applies for the *RelationshipConfiguration*. The integrity constraint records store *RelationshipTemplates* where each integrity constraint is described in it. Both *NodeTemplate* and *RelationshipTemplate* follow design steps that are shown in figure 3.4. Especially is necessary to mutually distinguish the kind of integrity constraints, thus we collect these sort of attributes such as a node label (or relationship type), the node (or relationship) properties, an integrity constraint name, an action (unique and exists possible clauses), the enable, validation, delete, update and final. Some of these attributes are not required for our purpose, but they might be useful in the future.

Representation structure of integrity constraints

The next task what is needed to have done is to create a definition (or schema) structure for storing defined integrity constraints in Cypher. We have chosen as a solution to use a JSON technology. It allows the stored structure also to pass defined database schema on demand of an integrity constraint management system. The database schema is illustrated by a structure defined as a JSON Schema. The JSON Schema is self-descriptive, and for determined integrity constraints is shown in listing 4.3. One of the many possible returned responses is shown in listing 4.4.

Listing 4.3: Integrity constraint structure - JSON Self-Descriptive Schema

```
{
  "$schema": "http://json-schema.org/schema#",
  "required": ["clause","name"],
  "properties": {
    "clause": {
      "type": "string",
      "description": "Clause identifier"
    },
    "name": {
      "type": "string",
```

```
      "description": "Name of the integrity constraint"
    },
    "pattern": {
      "type": "string",
      "description": "Pattern identifier"
    },
    "action": {
      "type": "string",
      "description": "Action identifier"
    },
    "properties": {
      "type": "string",
      "description": "Properties to be validated"
    },
    "options": {
      "type": "object",
      "properties": {
        "enable": {
          "type": "string"
        },
        "validation": {
          "type": "string"
        },
        "delete": {
          "type": "string"
        },
        "update": {
          "type": "string"
        },
        "final": {
          "type": "boolean"
        }
      }
    }
  }
}
```

Listing 4.4: Response example in a JSON structure

```
{
  "clause": "CREATE",
  "name": "MyICName",
  "pattern": "(node)-[TYPE]->(node)",
  "action": "EXISTS",
```

```
  "properties": "node.email",
  "options": {
    "enable": "VALIDATE",
    "validation": "DEFERRED",
    "delete": "STRICT",
    "update": "STRICT",
    "final": false
  }
}
```

Defining a graph database schema

To create a new schema for a graph database Neo4j, a *registerConfiguration* method is provided to register defined configurations for both node and relationship. If we have created either node configuration or relationship configuration, we can simply use the registerConfiguration method too with filled a null parameter for the node or relationship configuration. However, if we wanted to load defined integrity constraints stored in a .json file, then would be used a method taking a parameter *path* to pass the path where a .json file is physically stored on a disk for loading node or relationship database schema configuration. If you want to define multiple integrity constraints in one file you must separate each other with a newline to reach a proper process of these integrity constraints. For a manual configuration there are exist two separated possibilities the *NodeConfiguration* and *RelationshipConfiguration* for the manual definition of the integrity constraints in code. The method which provides this configuration flow is the *addNode* or addRelationship template which as an input takes an argument of the integrity constraint template definition in code. After the configuration load of .json file(s) (or manual definition of integrity constraints in code) is completed, then we must to inform the *SchemaConfiguration* environment by registering defined integrity constraints in a *registerConfiguration* method. If the register of our configurations were successful, we would be able to run an *enforce* method to start a database validation under the defined schema determined by the defined integrity constraints. Listings 4.5 and 4.6 describes a load operation of defined integrity constraints from the .json file, and manual definition of integrity constraints, respectively.

Listing 4.5: Schema definition from .json file

```
...

SchemaConfiguration schemaConfiguration =
   SchemaConfiguration.getInstance();
```

```
Configuration nodeConf = schemaConfiguration.
   configurationFactory.getConfiguration(
   ConfigurationType.NodeConfiguration);

nodeConf.loadNodeConfiguration("./schemaConfigurations/
   nodeConfigs.json");

schemaConfiguration.registerConfiguration(nodeConf,
   null);

...
```

Listing 4.6: Manual schema definition

```
...

SchemaConfiguration schemaConfiguration =
   SchemaConfiguration.getInstance();

Configuration nodeConf = schemaConfiguration.
   configurationFactory.getConfiguration(
   ConfigurationType.NodeConfiguration);

NodeTemplate constraintUser = new NodeTemplate("u:User",
    "u:email", "icUniqueUser", "unique", "validate",
    "deferred", "restrict", "restrict", false);

NodeTemplate constraintPerson = new NodeTemplate("p:
   Person", "p:username", "icUniquePerson", "unique",
   "validate", "deferred", "restrict", "restrict",
   false);

schemaConfiguration.registerConfiguration(nodeConf,
   null);

...
```

When creating a *NodeTemplate* or *RelationshipTemplate* we must fill arguments in a following sequence. A node label (or relationship type), the node (or relationship) properties, a name of your integrity constraint for further recognition in an integrity constraint management, a required action (the unique or exists), and the next arguments for enabling, validation time, delete and update behavior, and final behavior good for locking the whole data property changes.

Start schema enforcing of a graph database

As soon as we are done with the schema definition by the integrity constraints, and also we have already registered them in *SchemeConfigurationAPI* we are allowed to start the whole enforcement process. This can be done via calling a method *enforce* with provided transactionData and databaseService which is an instance of a *GraphDatabaseService* interface for keeping the instance of opened session of the current graph database. In **Neo4j Transaction Event API** 4.1 we introduced in listing 4.2 the use of transactions in Neo4j graph database. There are listed three implemented methods *beforeCommit*, *afterCommit* and *afterRollback*. It is highly recommended to start the schema enforcement at the *beforeCommit* method because there we have a privilege to validate data conditioned by the integrity constraints. The application is seen in listing 4.7.

Listing 4.7: Manual schema definition

```
...
GraphDatabaseService database;
SchemaConfiguration schemaConfiguration;
...
schemaConfiguration.registerConfiguration(...);
...
database.registerTransactionEventHandler(new
   TransactionEventHandler<Void>()
{
        @Override
        public Void beforeCommit(TransactionData
           transactionData) throws Exception
        {
                schemaConfiguration.enforce(
                   transactionData, database);
                // Other operations
                return;
        }
        @Override
        public void afterCommit(...) {...}
        @Override
        public void afterRollback(...) {...}
});
...
```

Delete and update database schema

A data update in Cypher is handled by MATCH and SET clauses. For our purpose the update operation is theoretically described in the design section 3. Due to our limited implementation we are not concerned in solving this statement practically, and it is simulated by manual data editing through iterator pattern. A delete operation is designed for a full schema removal or partial schema removal with the integrity constraint names included.

4.3 How to start using SchemaConfigurationAPI

The *SchemaConfigurationAPI* for a definition of a graph database schema by integrity constraints is implemented as a *Maven* project. Maven is a command line build tool used to build project and make the .jar files. If you wish to use this API in your own project there is required to use either to put a produced .jar file into the project classpath or clone the GitHub repository (https://github.com/JiriKovacic/constraints) and run *mvn clean install* and then produced the .jar file put into the your project classpath. Do not forget to check whether a pom.xml file must have included a GraphAware framework because some parts of it are in SchemaConfigurationAPI used (e.g. simple graph database shutdownhook or test methods from GraphUnit). If it has not specified there yet, the compilation would fail.

4.4 Summary

At the beginning of this realization chapter we introduced the crucial background such as Neo4j Core API and Neo4j Transaction Event API needed to start working with a Neo4j database instance. These API's also behave as a mediator to run the new implementation for creating a new schema definition for a graph database by integrity constraints. This prototype implementation is ready to add new types of integrity constraints and it is further expandable. There are already implemented three types of integrity constraints limited to nodes. It is **Node property uniqueness, Mandatory properties** and **Property value limitations**. We showed that is needed to have an instance of *SchemaConfigurationAPI*, and after defining the new integrity constraints we are able to enforce data in the graph database. What is hidden in the integrity constraint implementation itself is how to exactly perform the whole validation process. We did not mention it in the text above, but many problems had occurred. Neo4j Transaction Event API[4] provides methods for different situations which may arise. It is especially for the transaction data separation into classes named created, assigned and removed which applies for both nodes

[4] All operations within the transaction data require to use a provided Transaction interface, otherwise a transaction exception is thrown.

and relationships. He, who wants to work with the transaction data, must use these methods which are independent of each other. The next problem which was arisen was how to validate data in the database before they are actually fully loaded. This happens because there is not any way how to access the database through the transaction data. It is solved by provided an instance of the mapped database. All operations must be performed in a transaction, if we decide to check data in the database, we will read an uncommitted state of data which means that we can find out that a record exists, but we are not able to read the record content (i.e. property values). Thus, the solution which works is to create another transaction (called a nested transaction) in the first executing transaction and perform a commit action to data. This type of transaction is in Neo4j called **Placebo transaction**. The definition by Michael Hunger is as follows[50]: "Placebo transactions are nested transactions in Neo4j, they are created when there is already a top-level transaction running, and only affect the top-level transaction when they are:

- not finished,
- rolled back,
- terminated."

So, with use of placebo transactions we are able to make a partial commit and work with data as they were committed. But, if those data did not meet the criteria of defined integrity constraints, we could simply abort the top-level transaction and partially committed data would be rolled back.

Chapter 5

Measurements

The measurements chapter is focused on a total integrity constraint validation time during running process and how the integrity constraints affect a graph database when they are turned on. There is measured how much time is needed for defined integrity constraints to data enforcement process. We also focused on a validation of the full graph database because this database state represents the worst case which can occur and influence a net process and database performance. Data creation will be omitted; they have at most constant time complexity.

To test the graph database served us a **Cineasts** database[51]; possible nodes properties are taken from Hunger[52]. This dataset is aimed as a Movie database where used data come from TheMovieDB. The Cineasts database stores data about Movies, actors who played in a specific movie and their roles, users and given ratings for movies in their area of interest. The structure of the Cineasts database is shown in figure 5.1. Cineasts behaves a little bit as a social movie database because there is possible to make friends between database users. The database contains exactly 106 651 relationships and 63042 nodes, respectively.

For obtaining the most accurate results these measurements are always performed multiple times and results are averaged. There exist two metrics of measurement, the time and number of database hits. We measured only time

Table 5.1: Hardware configuration

Parameter	Description
Processor	Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz, 2301 Mhz, 4 Core(s)
RAM	8GB 1600MHz DDR3 SDRAM (2x 4GB)
HDD	1TB SATA (5400 RPM)
OS	Microsoft Windows 10 Pro (Build 10586)

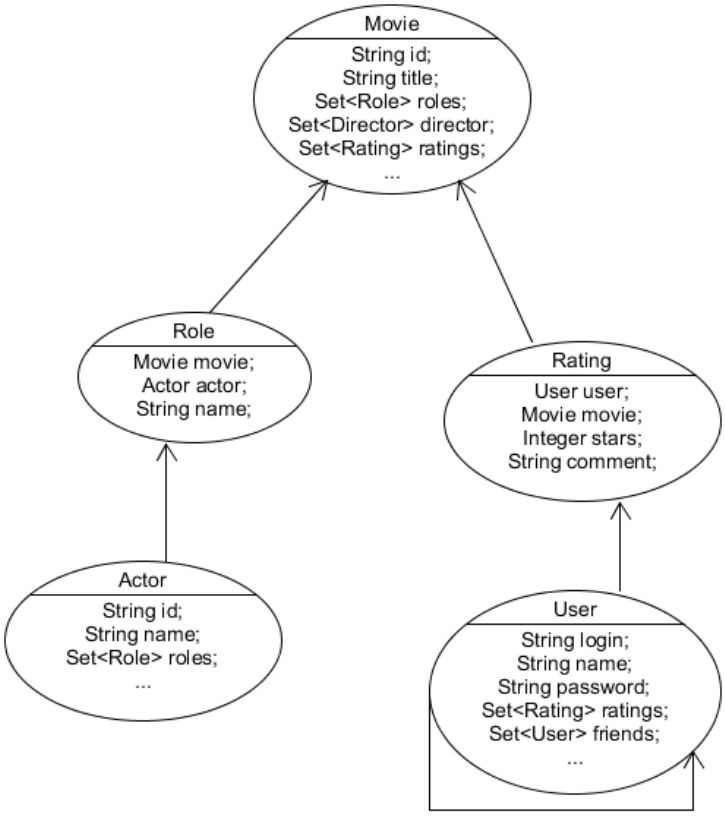


Figure 5.1: The structure of Cineasts database.

in milliseconds because the whole validation process in not performed in the database itself and it is not needed to measure database hits. For performing measurements is used a configuration shown in a table 5.1.

Above we mentioned that the Cineasts movie database stores around 63 thousand nodes. With use of a simple Cypher query *MATCH (n) RETURN n,count(*)* we can find out that therefrom 44 943 nodes stands for actors, 6 037 nodes for directors, 12 862 nodes for movies and 45 nodes for users. But, if we make a total sum of each node groups, we will see that the total number does not fit. This means that some records must be duplicated, and we are able to identify them by use of an integrity constraint called *Node property uniqueness*. As a reminder, it is good to know that complexity for the unique integrity constraint data enforcement is theoretically in the worst case $O(n^2)$. However, practically due to its implementation background it is higher. For the other implemented integrity constraints the time complexity is in theory the worst case $O(n)$ and only in the case if the non-valid data are placed at the end of the validation process.

Table 5.2: Node label segmentation

Cineasts nodes	Users	Directors	Movies	Actors
63887	45	6037	12862	44943

5.1 Characterization of measurements

All measurements were performed on the Cineasts database. There were measured three ways of defined integrity constraints for the database schema enforcement. All measurements show a slightly higher time because the integrity constraints are specified without using of some type of an implemented index on the node labels that would be indexed on which the integrity constraint is turned on. There are measured these following integrity constraints:

- Node property uniqueness - single property
- Node property uniqueness - multi property
- Property value limitations.

The measurement is based on how much time does it take to validate data across the whole graph database on which the integrity constraint defines its schema. Simply, in comparison with the relational databases, it is measured the state as *ENABLE VALIDATE* which means that the integrity constraint defined for the specific node label(s) starts the validation process of the whole graph database, and with time execution set on *IMMEDIATE* which means that the integrity constraints are started to control data by the database schema immediately. With this settings we achieve the validation of the whole graph database. Table 5.2 shows a representation of nodes at the Cineasts database. Node labels in that database cover Users, Directors, Movies and Actors with their partial sums in each segment, and a total sum listed in a *Cineasts nodes* table cell. For *unique* measurements are used only node groups with labels User which consist of 45 nodes, and a Director label consisting of about 6 000 nodes. For the *non-unique* measurement are used all nodes within the Cineasts movie database.

Measurement no. 1: Cineasts database – Unique single property validation

The unique single property validation is an integrity constraint where is for a specific node label defined a particular property which must be enforced in a graph database as a unique property. Definition in Cypher syntax looks as follows in listing5.1.

Listing 5.1: Node property uniqueness - Single property value

```
(1) CREATE CONSTRAINT (name:'uniqueUserName') ON
    (u:User) ASSERT UNIQUE(u.name) OPTIONS(enable:
    'VALIDATE', validation:'IMMEDIATE');

(2) CREATE CONSTRAINT (name:'uniqueDirectorName') ON
    (d:Director) ASSERT UNIQUE(d.name) OPTIONS(enable:
    'VALIDATE', validation:'IMMEDIATE');
```

Due to time consumption the integrity constraint for the single property uniqueness is decided to use nodes only with the *User* and *Director* labels. Above, in this chapter, we mentioned that measurements are performed multiple times and then the results are averaged. During the measurement process was database gradually modified to achieve a schema-approved (or integrity constraint approved) database state with no data violations. In figure 5.2 is demonstrated this fact as *Users or Directors with data violation* and *Users or Directors with NO data violation*. We can see that data with violations represents less time during the validation process then data with no violations. This is achieved by the integrity constraint implementation itself because when some data violate the integrity constraint rule, then the whole database validation process stops. The columns labeled with no violations represents the total validation time if we want to convince ourselves that data conform the specified database schema. This approach is very useful when we do first database initialization and we want to be ensured that data do not violate any database restrictions.

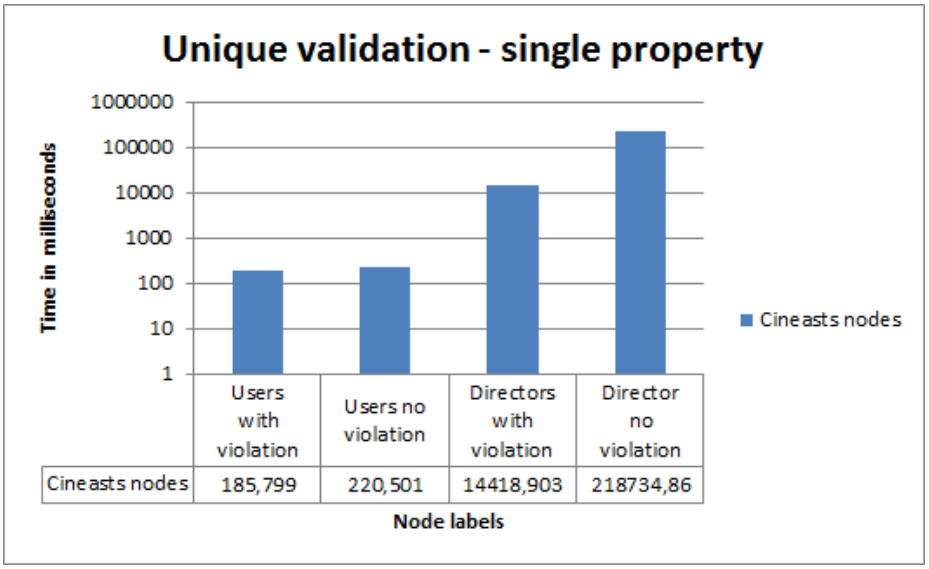


Figure 5.2: Unique validation – single property.

Measurement no. 2: Cineasts database – Unique multiple property validation

The next unique multiple property validation is the integrity constraint where is for a specific node label defined a particular combination of two properties, which must be enforced in the graph database, as the node unique properties. For example, if we define a new integrity constraint for a node with a label *Person* and properties with the *firstName* and *lastName*, then it is allowed to have nodes with the same firstName values as long as they do not have the same lastName value, and conversely the same lastName values as long as they do not have the same fistName value. Definition in the Cypher syntax for the Cineasts database example looks as follows in listing5.2.

Listing 5.2: Node property uniqueness - Multiple property value

```
(1a) CREATE CONSTRAINT (name:'uniqueUserComb') ON
     (u:User) ASSERT UNIQUE(u.id, u.name)
     OPTIONS(enable:'VALIDATE', validation:'IMMEDIATE');

(1b) CREATE CONSTRAINT (name:'uniqueUserComb') ON
     (u:User) ASSERT UNIQUE(u.id && u.name)
     OPTIONS(enable:'VALIDATE', validation:'IMMEDIATE');

(2a) CREATE CONSTRAINT (name:'uniqueDirectorComb') ON
     (d:Director) ASSERT UNIQUE(d.id, d.name)
     OPTIONS(enable:'VALIDATE', validation:'IMMEDIATE');

(2b) CREATE CONSTRAINT (name:'uniqueDirectorComb') ON
     (d:Director) ASSERT UNIQUE(d.id && d.name)
     OPTIONS(enable:'VALIDATE', validation:'IMMEDIATE');
```

As can be seen in listing5.2 there are implemented two ways of how to separate the property combinations to each other. For the Cineasts database was chosen a property *id* where is expected to be natively unique with a cooperation with a property *name*. Then the actual measurement itself was performed in the same way as the measurement for the single property unique validation 5.1.

Even here thanks to the time-consuming validation the integrity constraint is determined only for nodes with *User* and *Director* labels. Also here during the measurement process the database was gradually modified till it had been schema-approved. Figure 5.3 describes for the same nodes which are *Users or Directors with data violation* and *Users or Directors with NO data violation*. The next figure 5.4 shows a time-consumption comparison between those two single and multiple unique property validation. We can see that almost in all possible cases the multiple property value unique validation consumed more

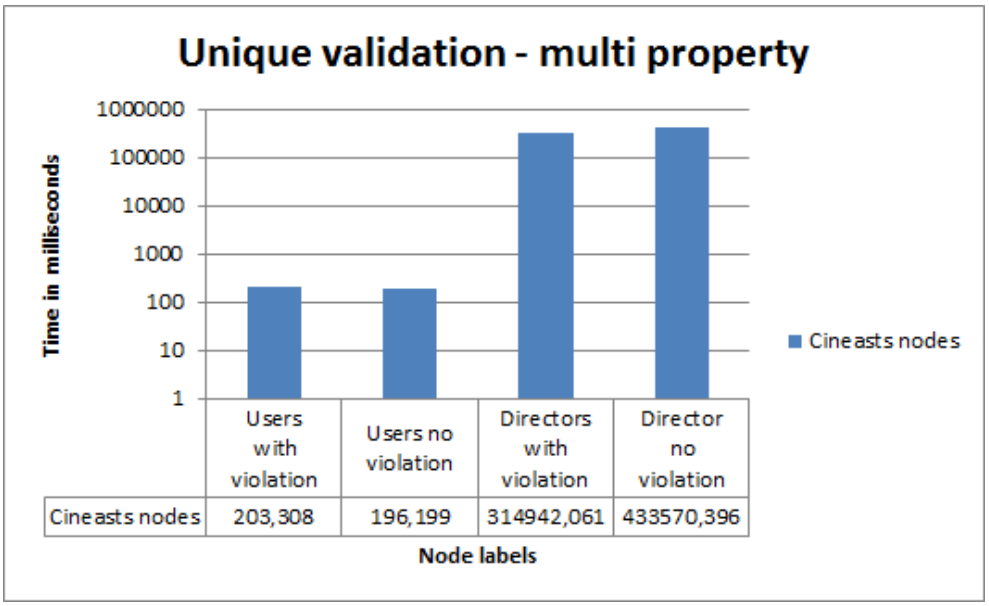


Figure 5.3: Unique validation – multiple property.

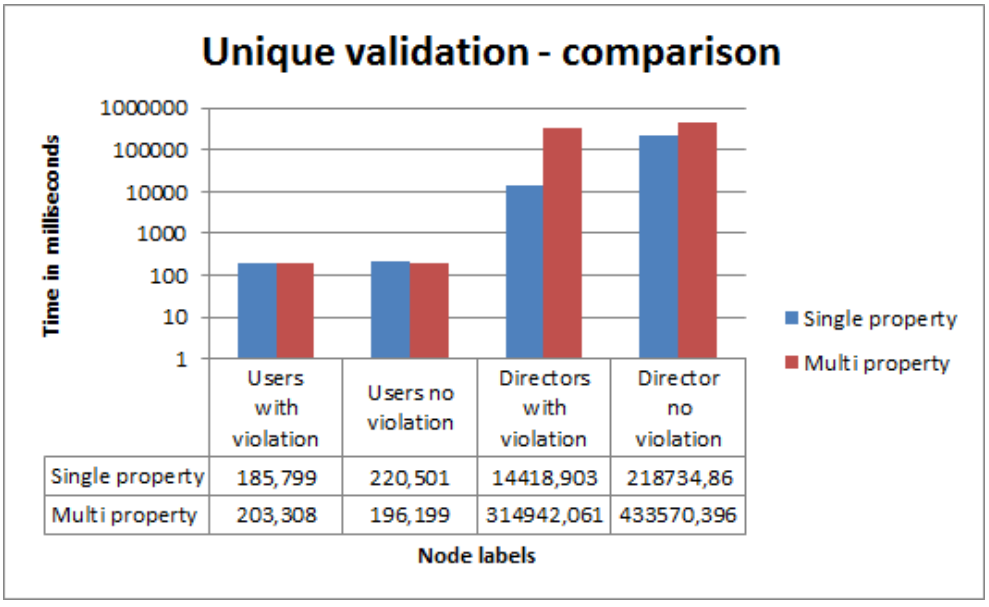


Figure 5.4: Unique validation – comparison.

time to check data in the database under the specified integrity constraint. The case "Users no violation" can be some kind of pathological cases. However, this can be more likely due to its small sample of data.

Measurement no. 3: Cineasts database – Property value limitations or Mandatory properties

The last one is a node validation called *Property value limitations*. The inner implementation body of this integrity constraint is very close to the *Mandatory property* integrity constraint and the measured results would be very similar. As an example the integrity constraint validates a STRING datatype for all nodes in the Cineasts database which consists of Users, Directors, Movies and Actors nodes. Following listing 5.3 shows us definitions in the Cypher language for the Property value limitations integrity constraint where is set the datatype STRING to the schema definition of the graph database.

Listing 5.3: Property value limitations - String datatype

```
(1) CREATE CONSTRAINT (name:'dtUser') ON (u:User) ASSERT
    EXIST(u.name AS STRING) OPTIONS(enable:'VALIDATE',
    validation:'IMMEDIATE');

(2) CREATE CONSTRAINT (name:'dtDirector') ON
    (d:Director) ASSERT EXIST(d.name AS STRING)
    OPTIONS(enable:'VALIDATE', validation:'IMMEDIATE');

(3) CREATE CONSTRAINT (name:'dtMovie') ON (m:Movie)
    ASSERT EXIST(m.title AS STRING) OPTIONS(enable:
    'VALIDATE', validation:'IMMEDIATE');

(4) CREATE CONSTRAINT (name:'dtActor') ON (a:Actor)
    ASSERT EXIST(m.name AS STRING) OPTIONS(enable:
    'VALIDATE', validation:'IMMEDIATE');
```

This validation process is less time-consuming than the unique integrity constraint because there is no need to validate data each other which requires many repetitive comparing operations. Figure 5.5 shows how much time is required to perform schema enforcement on the whole graph database. We may notice that there is a big difference between performance of those two Node property uniqueness and Mandatory property (or Property value limitations). Once again, it is due to that the unique validation requires $O(n^2)$ steps and property value limitation requires only $O(n)$ in algorithmic complexity.

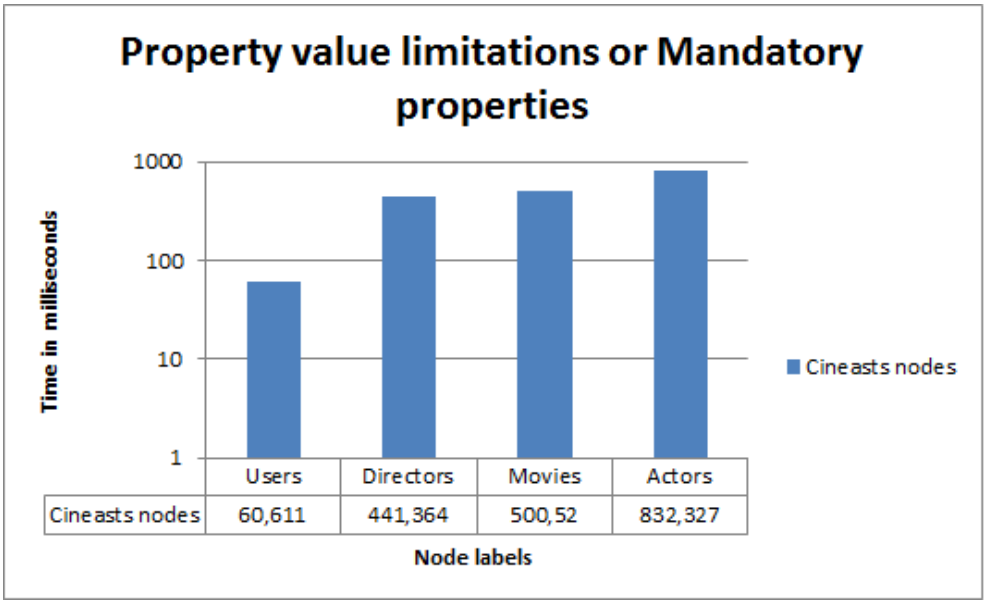


Figure 5.5: Property value limitations or Mandatory properties.

Overall evaluation

We have performed three types of measurements of defined selected integrity constraints. These integrity constraints were *Node property uniqueness*, *Mandatory property* and *Property value limitations*. The measurements were conducted on the Cineasts movie graph database which consist of more than 63 thousand nodes and over 100 thousand relationships. The whole prototype implementation was intended to validate only nodes, but the whole implementation design is ready to add the new functionalities for relationship validation, and also for a definition of the new integrity constraints. The performed measurements itself shows that it is possible to use a database schema specification declared by the integrity constraints and apply the schema on the database for data validation.

We found out that it is in our power to use those integrity constraints. What makes a little bit problems is the unique validation. The problem can occur while we must enforce the database schema. This can happen during database initialization and data load. The ability to control and validate data is at the first time very time-consuming. However, we can argue that this is done once, and then the further loading of data does not take so much time. This is happening by the implementation itself which is divided into three states. For nodes those states are *created*, *assigned* and *removed*. These states are independent each other and accelerates the validation process.

Unfortunately in measurements we were not able to compare prototype integrity constraint implementation with the current implementation in the

Neo4j graph database. In Neo4j, for example, we are not able to execute validation process to the whole database because there is not implemented such functionality and we could not compare our obtained results with the current integrity constraint implementation, especially with the unique integrity constraint. If we talk about the unique integrity constraint in Neo4j and what nowadays is provided, then it is a definition of unique integrity constraint with a validation control for newly incoming data and that's all.

The next other implemented integrity constraints are convenient and the measurements above this conformed. For the control of the whole database containing more than 63 thousand nodes the validation process took less than 1 second which is very good outcome.

In view of the fact that in the current Neo4j implementation the integrity constraint after its creation is automatically accompanied by the implementation of some kind of index to make less cost querying. This fact can speed-up the validation process itself, but an implementation of some kind of index was not intented in this thesis.

Chapter 6

Future work

This diploma thesis gently broached the issue of definition integrity constraints in Cypher Query Language. The work that must be done around this topic is very widespread, and it requires more concentration that is reserved. The integrity constraints problem is a very extensive theme. At the beginning we were concerned at how the integrity constraints are processed in the relational databases such as the Oracle and PostgreSQL database. We were inspired from many ideas that are common in the relational databases. These ideas are implemented, tested, and they work well in the relational databases, so why not to inspire from them and use their functionality in the Neo4j graph database. The study involved seven different cases (or types) of integrity constraints. In revision chapter 2.3 became clear that some types of integrity constraints are not convenient for example *Endpoint requirements*.

The implementation itself acts as a prototype API for an embedded mode of the Neo4j graph database. There are implemented three types of integrity constraints which are concerned only for a node validation which work pretty well. The whole implementation is proposed to make an easy addition of new functionalities such as integrity constraint validation for relationships. There is also possible to create new methods for implementation of other integrity constraint types. The next integrity constraints that could be implemented are *Required relationships, Cardinality requirements* or *Label coexistence*. However, we must bear in mind that the validation process takes a constant time, but if we had to validate many node relationships multiple times, we would obtain many repetitive comparing operations and we would lose the performance. In those cases should be helpful to use some heuristics if they exist.

The prototype implementation does not use indexes. The next what should be implemented is a creation of index for a specific node label (or relationship type) at the time when the integrity constraint is created. Other thing at the integrity constraint definition is a pattern requirement, thus what we want to validate. For *Node property uniqueness, Mandatary properties* and *Property value limitations* integrity constraints is only needed a pattern in a

form of the node representation like *(n:NodeLabel)*. However, the next possible integrity constraints can be defined with these kind of patterns *()-[]-()* or long-path patterns for special integrity constraints like *()-[]-()-[]-()* etc[5]. So the future work would deal with these patterns too.

The next what should be implemented is support of the Neo4j graph database at a stand-alone mode. This means that provided Cypher, in which the integrity constraints are defined, through an exposed web interface and use of REST API should be transferred to the database where the particular integrity constraint definition would be processed, and the inadequate Cypher would be forwarded further. The extension of the prototype implementation could lie as a plugin which would enrich the actual installation of the Neo4j graph database where the plugin would be placed into the appropriate Neo4j plugin folder.

If we take a look at Cypher in which the integrity constraints are defined we can imagine that the implementation should contain some kind of Cypher parser or using of the recommended solution which is currently in use by Neo4j. The solution is available at a Neo4j blog [53] and the actual execution plan is as follows:

- Convert an input query string into an abstract syntax tree (AST).
- Optimize and normalize the AST.
- Create a query graph from the normalized AST.
- Create a logical plan from X.
- Rewrite the logical plan.
- Create an execution plan from the logical plan.
- Execute the query using the execution plan.

There exists so much work what to do and what to implement for the Neo4j graph database. The presented problems would be covered in several following diploma thesis which can bring on to the light for the new possibilities and ideas.

[5]Parentheses () represents nodes and brackets [] represents relationships

Conclusion

Graph databases are one of the fastest growing databases systems in the world. Graph databases are much younger than relational databases and in some ways they are taking a little bit of inspiration from them because of well-established processes. Such a case involves this diploma thesis as well. Before concerning of the proposals of the integrity constraints there must be conducted an appropriate research where should have covered the most important and useful methods that are common for the world of the relational databases because in the *Request for proposals: Cypher Schema syntax* paper suggested by *Cypher language group* is not covered this possibility. Integrity constraints are such defined rules that describe a database schema. Graph databases including Neo4j are schema-free and the already implemented integrity constraints are not as good as they should be. The integrity constraint term can be represented for example that some data in the graph database must be unique etc.

We have laid the foundations for a new Cypher Query Language syntax for creating integrity constraints which define a certain database schema needed to be enforced. During the work we have introduced all possible integrity constraints that are proposed on RFP paper. These integrity constraints are *Node property uniqueness*, *Mandatory properties*, *Property value limitations*, *Required relationships*, *Cardinality requirements*, *Endpoint requirements* and *Label coexistence*. The major contribution is in the **Revision chapter** where those integrity constraints from several points of view are discussed. Primarily we were focused on **Syntax** and how the Cypher syntax for integrity constraints should look like, after that **Usefulness** and which the integrity constraint deals with, **Complexity** where we had to introduce problems and possible difficulties with a combination of a selection of a graph database validation mode and DML operations such as insert, update and delete, and last but not least it is needed to deal with postponing of a validation process for the cases if we want to validate data in an immediate or deferred mode. When the theory and discussion was done it came turn to design the new Cypher Query

Language syntax for proposed new integrity constraints. The CQL syntax itself was designed gently to keep the basic idea of the actual Cypher syntax for querying the graph database. The designed Cypher syntax covers all possible options from creation of integrity constraint, its detection by Cypher MATCH clause, modifications by SET clause and there are also provided possible actions to drop, enable and disable these integrity constraints.

The implementation was intended as a prototype. The implementation is designed for all kinds of integrity constraint, but supports a subset of them and is targeted only for the node validation. Exactly, there are supported integrity constraints *Node property uniqueness, Mandatory properties* and *Property value limitations*. The implementation exposes its interface which can be used with JVM-language. At first, it is needed to create an instance of *SchemaConfiguration* which registers divided configurations for both nodes and relationships. Configurations for nodes can be defined in-code or in-file where for description of defined integrity constraint is used a JSON structure format. After registering the sub-configurations just need to run enforcement process and the rest of the internal implementation takes care of everything. Also the prototype implementation is available at GitHub repository where the source code is available.

The implemented integrity constraints were tested at the movie database called Cineasts which consists of around 63 thousand nodes and over 100 thousand relationships. We found out that implemented integrity constraints can exist and should be included into Neo4j graph database. There were appeared some difficulties with *Node property uniqueness* integrity constraint and its time consumption at the first validation of initialized graph database state. The Unique validation is a very expensive operation and it is become the more expensive, the more data in the database must be checked. However, this full database scan happens only at the first time and primarily depends on a user and how much he trusts the data because there is a possibility to switch off the validation control.

The thesis aims to introduce such possibilities how to behave towards integrity constraints from several points of view. The main part was to focus at the theory and possible difficulties that may arise. After that design a suitable Cypher syntax for integrity constraint definition followed an implementation of a prototype program to demonstrate proposed functionalities. However, there is still much work that is included at this topic.

Bibliography

[1] TutorialsPoint. DBMS - Overview [online]. [cit. 2016-04-27]. Available from: http://www.tutorialspoint.com/dbms/dbms_overview.htm

[2] Chong, R. F.; Wang, X.; Dang, M.; et al. *Understanding DB2: Learning Visually with Examples (2nd Edition)*. IBM Press, 2008, ISBN 0131580183. Available from: http://www.ibmpressbooks.com/articles/article.asp?p=1163083

[3] Troup, M. *Indexing of patterns in graph DB engine Neo4j I*. Master's thesis, Czech Technical University in Prague, 2015.

[4] Alexanderson, G. L. About the cover: Euler and Königsberg's bridges: a historical view. *Bull. Amer. Math. Soc. (N. S.)*, volume 43, no. 4, 2006.

[5] Tutte, W. T. *Graph Theory (Cambridge Mathematical Library)*. Cambridge University Press, 2001, ISBN 0521794897.

[6] Paoletti, T. Leonard Euler's Solution to the Königsberg Bridge Problem [online]. [cit. 2016-03-18]. Available from: http://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem

[7] Diestel, R. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2000, ISBN 0387989765.

[8] Bachman, M. *GraphAware: Towards Online Analytical Processing in Graph Databases*. Master's thesis, Imperial College London, 2013.

[9] New South Wales, U. Methods of Search [online]. [cit. 2016-02-25]. Available from: http://www.cse.unsw.edu.au/~billw/Justsearch.html

[10] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; et al. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001, ISBN 0262032937.

[11] Date, C. *An Introduction to Database Systems (8th Edition)*. Pearson, 2003, ISBN 0321197844.

[12] Marzi, M. D. Introduction to Graph Databases [online]. [cit. 2016-02-25]. Available from: http://www.slideshare.net/maxdemarzi/introduction-to-graph-databases-12735789?related=1

[13] Hunger, M. Intro to Graphs and Neo4j [online]. [cit. 2016-02-25]. Available from: http://www.slideshare.net/neo4j/intro-to-graphs-and-neo4j-33677154?next_slideshow=2

[14] MongoDB. Document-oriented database [online]. [cit. 2016-02-26]. Available from: https://www.mongodb.com/document-databases

[15] Robinson, I.; Webber, J.; Eifrem, E. *Graph Databases*. O'Reilly Media, Inc., 2015, ISBN 9781491930892.

[16] Neo4j. The Neo4j Graph Database [online]. [cit. 2016-02-28]. Available from: http://neo4j.com/docs/stable/graphdb-neo4j.html

[17] Bell, P.; Hunger, M. Get more intelligence from your data using Cypher and Neo4j – UPDATED for Neo4j 2.0 [online]. [cit. 2016-02-27]. Available from: https://jaxenter.com/get-more-intelligence-from-your-data-using-cypher-and-neo4j-updated-for-neo4j-2-0-107498.html

[18] Neo4j. Chapter 1. Neo4j Highlights [online]. [cit. 2016-02-29]. Available from: http://neo4j.com/docs/stable/introduction-highlights.html

[19] Michon, C. H2 database - embedded or server mode? [online]. [cit. 2016-02-29]. Available from: http://stackoverflow.com/questions/26418682/h2-database-embedded-or-server-mode

[20] Neo4j. Chapter 25. High Availability [online]. [cit. 2016-02-29]. Available from: http://neo4j.com/docs/stable/ha.html

[21] Neo4j. 8.1. What is Cypher? [online]. [cit. 2016-03-01]. Available from: http://neo4j.com/docs/stable/cypher-introduction.html

[22] Freeman, W.; Needham, M. Optimizing Cypher Queries in Neo4j [online]. [cit. 2016-03-01]. Available from: http://www.slideshare.net/neo4j/optimizing-cypher-32550605

[23] Neo4j. 14.2. Constraints [online]. [cit. 2016-03-01]. Available from: http://neo4j.com/docs/stable/query-constraints.html

[24] Bachman, M. Introducing GraphAware Neo4j Framework [online]. [Cited 2016-03-26]. Available from: http://graphaware.com/neo4j/2014/05/28/graph-aware-neo4j-framework.html

[25] Bachman, M. GraphAware Neo4j Framework [online]. [Cited 2016-03-26]. Available from: https://github.com/graphaware/neo4j-framework

[26] Neo4j. 32.1. Server Plugins [online]. [cit. 2016-03-27]. Available from: http://neo4j.com/docs/stable/server-plugins.html

[27] Bachman, M. Neo4j Improved Transaction Event API [online]. [Cited 2016-03-27]. Available from: http://graphaware.com/neo4j/transactions/2014/07/11/neo4j-transaction-event-api.html

[28] Oracle. Defining Tables [online]. [cit. 2016-03-03]. Available from: https://docs.oracle.com/cd/E18315_02/doc.214/e18306/tables.htm

[29] Valenta, M. Integritní omezení [online]. [cit. 2016-03-03]. Available from: https://users.fit.cvut.cz/valenta/doku/lib/exe/fetch.php/bivs/dbs2_02_io_ddl.pdf

[30] Bílek, P. Sloupcová omezení, Indexy [online]. [cit. 2016-03-03]. Available from: http://www.sallyx.org/sally/psql/psql5.php

[31] Vebloud. Teorie relačních databází: Integritní omezení [online]. [cit. 2016-03-03]. Available from: http://www.manualy.net/article.php?articleID=15

[32] PostgreSQL. Constraints [online]. [cit. 2016-03-03]. Available from: http://www.postgresql.org/docs/9.5/static/ddl-constraints.html

[33] Oracle. CONSTRAINT clause [online]. [cit. 2016-03-03]. Available from: https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqlj13590.html

[34] PostgreSQL. Foreign keys [online]. [cit. 2016-03-04]. Available from: http://www.postgresql.org/docs/current/static/tutorial-fk.html

[35] Kozubek, A. ON DELETE RESTRICT vs NO ACTION [online]. [cit. 2016-03-04]. Available from: http://www.vertabelo.com/blog/technical-articles/on-delete-restrict-vs-on-delete-no-action

[36] TechOnTheNet. ORACLE/PLSQL: FOREIGN KEYS WITH CASCADE DELETE [online]. [cit. 2016-03-04]. Available from: http://www.techonthenet.com/oracle/foreign_keys/foreign_delete.php

[37] Consulting, B. Enabling and Disabling Constraints [online]. [cit. 2016-02-14]. Available from: http://www.dba-oracle.com/t_enabling_disabling_constraints.htm

[38] Oracle. CONSTRAINT clause [online]. [cit. 2016-02-14]. Available from: https://docs.oracle.com/cd/B28359_01/server.111/b28310/general005.htm

[39] Foote, R. NOVALIDATE Constraints – No really ... [online]. [cit. 2016-02-14]. Available from: https://richardfoote.wordpress.com/2008/07/28/novalidate-constraints-no-really/

[40] Pedersen, A. A. Enabling and disabling Oracle Constraints [online]. [cit. 2016-02-14]. Available from: http://www.databasedesign-resource.com/enabling-and-disabling-oracle-constraints.html

[41] PostgreSQL. ALTER TABLE [online]. [cit. 2016-02-14]. Available from: http://www.postgresql.org/docs/9.5/static/sql-altertable.html

[42] PostgreSQL. SET CONSTRAINTS [online]. [cit. 2016-02-14]. Available from: http://www.postgresql.org/docs/9.5/static/sql-set-constraints.html

[43] Davey, J. Deferring constraints in PostgreSQL [online]. [cit. 2016-02-14]. Available from: https://hashrocket.com/blog/posts/deferring-database-constraints

[44] Govind. Deferred Constraints [online]. [cit. 2016-02-14]. Available from: http://myorastuff.blogspot.cz/2009/05/deferred-constraints.html

[45] Oracle. Deferred Constraint Checking [online]. [cit. 2016-02-14]. Available from: https://docs.oracle.com/cd/B19306_01/server.102/b14220/data_int.htm

[46] Lindaaker, T. Request for proposals: Cypher Schema syntax. [cit. 2016-03-15].

[47] Tutorialspoint. DBMS - Data Schemas [online]. [cit. 2016-03-27]. Available from: http://www.tutorialspoint.com/dbms/dbms_data_schemas.htm

[48] GenMyModel. Room reservation schema [online]. [cit. 2016-03-28]. Available from: https://repository.genmymodel.com/online-example/room-reservation-schema

[49] Neo4j. Terminology [online]. [cit. 2016-03-27]. Available from: http://neo4j.com/docs/stable/terminology.html

[50] Hunger, M. How Spring Data Neo4j does choose the type of the transaction (Placebo or TopLevel) [online]. [cit. 2016-04-19]. Available from: http://stackoverflow.com/questions/24649646/how-spring-data-neo4j-does-choose-the-type-of-the-transaction-placebo-or-toplev

[51] Misquitta, L. Full dataset (12k movies, 50k actors) of the Spring Data Neo4j Cineasts.net [online]. [cit. 2016-04-20]. Available from: http://neo4j.com/developer/example-data/

[52] Hunger, M. Tutorial Spring Data Graph [online]. [cit. 2016-04-20]. Available from: https://github.com/neo4j-examples/sdn4-cineasts/wiki

[53] Neo4j. Introducing the new Cypher Query Optimizer [online]. [cit. 2016-04-26]. Available from: http://neo4j.com/blog/introducing-new-cypher-query-optimizer/

Appendix A

Acronyms

ACID Atomicity, Consistency, Integration, Durability

BFS Breadth-first search

CE Community Edition

CRUD Create, Read, Update, Delete

CQL Cypher Query Language

DBMS Database Management System

DDL Data Definition Language

DFS Depth-first Search

DML Data Manipulation Language

EE Enterprise Edition

GDBMS Graph Database Management System

IC Integrity Constraint

JSON JavaScript Object Notation

NoSQL Not only Structured Query Language

RFP Request for proposals

SQL Structured Query Language

Appendix **B**

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `measurements` ..... measured results from chapter 5
- `src` ..... the directory of source codes
  - `impl` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format