MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

Defining referential integrity constraints on NoSQL datastores

Masson, Thibaud; Ravet, Romain

Award date:
2019

Awarding institution:
University of Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

Defining referential integrity constraints on NoSQL datastores

Thibaud MASSON Romain RAVET

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Université de Namur
Faculty of Computer Science
Academic Year 2018–2019

Defining referential integrity constraints on NoSQL datastores

Thibaud MASSON Romain RAVET

Internship mentor: Francisco Javier BERMÙDEZ RUIZ

Supervisor: ______________________ (Signed for Release Approval - Study Rules art. 40)
Anthony CLEVE

A thesis submitted in the partial fulfillment of the requirements
for the degree of Master of Computer Science at the University of Namur

Acknowledgements

This thesis is completely based on a three and a half months internship at the University of Murcia in Spain, where we were welcomed in the research laboratory. The Computer Sciences Faculty at this university contains some researchers who are making interesting works on NoSQL datastores with tools like Neo4J and MongoDB.

We would like to sincerely thank our supervisor Anthony CLEVE and our sending institution, the University of Namur, for giving us the possibility to live this grateful experience in a foreign country, for coming to visit us as well as giving us many advice and a precious feedback about the work.

We would like to also thank our internship mentor, Francisco Javier BERMÚDEZ RUIZ for his work expertise, his guidance and his help to integrate us into the social life of this new city by doing regular activities.
Our acknowledgements also go to Jesús GARCÍA MOLINA and Diego SEVILLA RUIZ for the interesting meetings that we had and the precious advice that they were able to offer us.

Finally, we would like to thank our respective families, friends and fellow students for their support during all the duration of this work.

Abstract

Nowadays, information systems have to respond to needs which become more and more complex. Therefore, they become more difficult to use and have to take into account a bigger data amount. This is why a lot of people are using NoSQL datastores now, allowing a better flexibility during the development than traditional relational databases.
However this kind of database does not handle the data integrity, as well as the Referential integrity constraints. This is why the idea of defining and validating these Referential integrity constraints would be a good step forward in the context of the NoSQL datastores.
On a technical point of view, this referential constraints managing system could take advantage of the model-driven techniques. Model-driven engineering (MDE) enables developers to build code generative architectures by means of model transformations. Specifically, the implemented solution would be a language representing the definition and the validation of referential integrity constraints. With that, a code generation tool will also be developed. It remains important to point out that although this work is confined to the graph-oriented NoSQL datastores, the language and its supporting is oriented to any NoSQL source.

Keywords : NoSQL datastores, Graph-oriented, Referential Integrity constraints, Model-driven engineering techniques, Code generation

Résumé

De nos jours, les systèmes d'information doivent répondre à des besoins qui ne cessent de se complexifier. Ils deviennent donc plus difficile à utiliser et doivent prendre en compte un plus grand nombre de données. C'est pourquoi à l'heure actuelle, beaucoup de personnes ont recours aux bases de données NoSQL permettant une meilleure flexibilité lors du développement que les traditionnelles bases de données relationnelles.
Cependant, le problème de ce type de base de données est qu'elle ne gère pas l'intégrité des données. C'est pourquoi, l'idée de définir et de valider des contraintes d'intégrité référentielle serait une avancée intéressante dans le cadre des bases de données NoSQL.
D'un point de vue technique, ce système de gestion des contraintes référentielles pourrait tirer parti des techniques d'ingénierie dirigée par les modèles. L'Ingénierie dirigée par les modèles (IDM) permet aux développeurs de construire des architectures génératrices de code au moyen de transformations de modèles. Plus précisément, la solution mise en oeuvre serait un langage représentant la définition et la validation des contraintes d'intégrité référentielle. Avec cela, un outil de génération de code sera également mis au point. Bien que ce travail soit limité aux bases de données NoSQL orientées graphiques, le langage et son support sont orientés vers n'importe quelle source NoSQL.

Mots-clés : Bases de données NoSQL, Orienté-graphe, Contraintes d'intégrité référentielles, Techniques d'ingénierie dirigée par les modèles, Génération de code

Contents

List of Figures

Chapter 1

Introduction & Motivation

Since the 1970s, information systems depended on the relational databases. Nowadays, they become more and more complex and need additional flexibility to manage an increasingly amount of data. Therefore, the non-relational databases become more popular within the actual information systems due to their heterogeneity and their ability to easily adapt themselves. They save data without explicit structure which has the advantage of a better flexibility on this structure during the development process.

1.1 NoSQL Datastores

This thesis relies on several technical concepts concerning the non-relational databases, so it is firstly due to explain this notion. Although in a relational database, a schema has to be clearly defined before any information addition, it is not the case for non-relational database.

The focus of this work will be on NoSQL datastores that provide a better performance and flexibility for the modern applications. *Datastores* is the term referring to this kind of database, while the *Database system* is the term used for traditional databases.

The non-relational databases are classified into four distinct types :

- **Key-Value Store Databases** : Every item is stored as a key and has a value.
- **Document-Oriented Databases** : Link every item with a complex data structure.
- **Column Store Databases** : Save the data as columns to optimise queries.
- **Graph-Oriented Databases** : Uses nodes and relationships to represent stored data.

Among these different types of non-relational databases, *Graph-oriented databases* are the ones that have been chosen for this work.

1.2 Graph-oriented Datastores

Graph-oriented datastores can be used to represent references in a simple way and which would be more difficult to model in the relational paradigm. This kind of database uses nodes, edges and properties to build a graph representing the stored data.

The data items are represented by nodes with properties and edges where properties correspond to their attributes and edges correspond to their relationships with other nodes, as shown on the Figure 1.1.

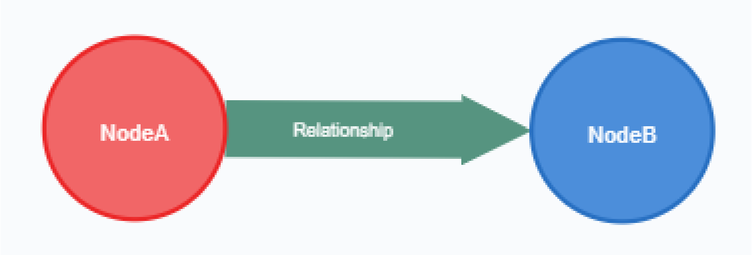


Figure 1.1: Data representation in Neo4J

In our case, Graph-oriented databases are accessed through APIs in the Java code. We have targeted two APIs in this work that are *Neo4J Java Driver* and *JCypher*. These two APIs will be described in more details in the section 2.3.1 and will be compared later on.

1.3 Referential Integrity Constraints

A Referential Integrity Constraint (RIC) is a rule that will ensure that the references and relationships between the elements are correct. In other words, it prevents users or other applications from impacting negatively the data and the structure within the DBMS.

The aim is to ensure consistency within the database when several tables have relationships between them, like a link *Foreign Key - Primary Key*. A table cannot be modified or deleted, for example, without impacting the other tables related to it. A RIC brings a solution to the problem of cascading actions.

A simple example would be : in a situation where an author is linked with a book, we could write a Referential integrity constraint which would prevent to delete this author as long as he would have a reference with at least one book.

1.4 Motivation of the work

As we said previously, non-relational datastores are emerging and it can be noticed that this kind of database does not explicitly handle with the data integrity. This difference compared to relational databases may discourage some people from using them. That is why this work aims to find an interesting

solution to handle with the data integrity in NoSQL datastores.
Using Referential Integrity Constraints could improve the consistency of the data and therefore its integrity.

1.5 Thesis Contribution

This thesis aims to develop a solution that manage Referential integrity constraints in a non-relational database context. Most of the time, RICs are used with relational database but rarely in a non-relational context. Therefore, we will explore this possibility as non-relational database become prominent.

The approach used in this work aims to use Model-Driven engineering techniques such as creating a Domain-Specific Language and generating code through a Model-To-Text transformation.

Our process for this work is the following :

- First, establish the research domain and design a metamodel that corresponds to the structure of a RIC. Then, a grammar can be defined with a DSL following this metamodel.
- After that, it is necessary to create some examples of RICs that will be save in a model (serialized in a XMI file).
- The next step is to extract the data from the XMI file and to apply on them a Model-To-Text transformation to get the Java code.
- Finally, all the remains is to valid each RIC, one by one, by executing the code generated and to check in the Graph-oriented database, managed by Neo4J, if the relationship is valid or not.

The developed solution will provide an editor tool supporting the DSL and will enable the RIC validation.

1.6 Model-Driven Engineering concepts

Model-Driven Engineering (MDE) uses models to improve software productivity and some aspects of software quality such as maintainability and interoperability. The advantage is that models and metamodels provide a high level formalism with which to represent artefacts.
Our solution will be based on two techniques of Model-Driven Engineering (MDE) that we were able to explore in depth thanks to the book [2].

The two MDE techniques applied in this work are :

1. The creation of a **Domain-Specific Language** (DSL) for defining the grammar of Referential integrity constraints on NoSQL datastores.
2. The **Model-To-Text** (M2T) transformation to generate the code that will provide the semantic of the previous language.

1.6.1 Domain-Specific Language

Domain-Specific Languages (DSL) are made to define the abstract and concrete syntax of a language. Then, some techniques allow to create the semantic of the developed language. A DSL offers appropriate constructs and notations to solve issues in a particular application domain, it can be implemented in three different methods, like it is explained in the slides of our host institution [18].

Firstly, the **external DSLs** which are created from scratch and there is needed to build a parser and an interpreter.
Then, the **internal DSLs** that are represented in two categories : Embedded DSLs and Fluent API.
At last, the **DSL workbenches**, also known as "Metamodel-based tools". For this case, a parser, an injector, a generator and an editor are directly generated from a specification based on a metamodel.

1.6.2 Model-To-Text Transformation

The second part of the work is intended to provide the semantic of the language by means of a code generation through a M2T transformation language. Starting from a basic model, the purpose is to obtain an executable code directly usable. We can pinpoint several benefits :

- Static and dynamic code separation by using a template-based approach to develop M2T transformations
- Explicit output structure
- Declarative request language
- Reusable basic feature

1.7 Thesis Structure

The chapter 2 is about the **Background** related to this work. Indeed, this Chapter will be divided in three parts, the first one being about the **Related Work** linked with the previous works made in the context of the thesis, the second part will be about the **Conceptual Background** needed. While the technologies that are used in this work will be explained in the part **Technical Background**.

The Chapter 3, **Methodology**, explains the approach of our solution and details the choices made during this project in accordance with the internship supervisor to conduct this work on a proper way.

The Chapter 4 presents the **Design** and the conception of a Referential integrity constraint. It is related to the different structural choices made and their evolution during the work.

The **Implementation**, Chapter 5, describes all the code implementation made to lead to the created system.

The Chapter 6 describes the **Taxonomy of Queries** by using *JCypher API* and *Java Driver API 1.7*. The goal will be to explain how queries are constructed according to these two languages.

A set of tests has been defined in the Chapter 7, **Experiments**, covering most of the possibilities leading to RIC validation. The following code generation and the behaviour of the selected graph-oriented will also belong to this chapter.

A **Conclusion**, Chapter 8, summarises this thesis and refers to potential issues that we have faced.

And finally Chapter 9 is dedicated to several thoughts about the possibilities of **Future Works** in this area and about what has not been achieved during the **Implementation** (Chapter 5) according to the **Design** (Chapter 4).

Every resource linked with the Bibliography or the Appendix part at the end of this manuscript will be strictly referenced.

Chapter 2

Background

The purpose of this chapter will be to explore useful technologies and several works already accomplished in a similar context to this thesis. It will be divided into three parts : the first one will be devoted to the related work, the second will aim to address conceptual information such as Referential integrity constraints, and finally, the last one will focus on technical topics such as the tools and methods used.

2.1 Related work

This first part will have the objective to talk about the previous works dealing with a subject close to this one. Different relevant articles and thesis will be explained here.

The first work to analyse is a thesis dating from 2018 [10]. This thesis aims to provide solutions to facilitate the implementation of lost implicit foreign keys. However, that work focus itself on relational databases while the actual one is focusing on non-relational databases. The concept of foreign keys is similar to our thesis, which consists of focusing on references between the elements.

Another thesis is helpful here, talking about referential integrity in cloud NoSQL datastores [15]. This thesis aims to present an API providing the CRUD operations to perform on the Database Management Systems ensuring that RICs are satisfied. That work also deals with the design of RIC in NoSQL datastores, while the focus is on Key-Value Store Databases (Section 1.1). The results are oriented about the performance of the CRUD operations under RICs.

Finally, the article [9] explains what “Referential integrity” is and the possibility of using the MapReduce algorithm within them. That work is made for document-oriented databases, but not for the graph-oriented databases. These kinds of NoSQL datastores are presented in the section 1.1. The main goal is to detect incorrect references in Document-oriented datastores due to mistakes in the code or in the transactions. For “One-To-Many” relationships, it checks the relationship integrity. This looks like at an approach that was used to implement the *Join* operations in NoSQL datastores.

2.2 Conceptual background

This section is related to the theoretical concepts that have to be explained in order to improve our understanding before continuing in the thesis. We are going to start by addressing the subject of Integrity Constraints.

2.2.1 Integrity Constraints

In the first place, we will approach the Integrity constraints in a non-relational context with a first paper which is close enough to the subject of our thesis. The article [21] will focus on the Integrity constraints in Graph-oriented databases.

As we know, Graph-oriented databases brings more flexibility and an easier schema evolution that relational databases. However, this has a negative impact on the consistency of the data and its management. This can also have a direct impact on the implementation of Integrity constraints which is more difficult.
The purpose of these constraints will be to bring consistency to these databases. Various kinds of Integrity constraints are developed in the article to maintain the consistency :

- Schema-instance consistency
  - Verifies the completeness and existence of data before insertion.
- Data redundancy
  - Sort through the information recorded in the database to remove those that are redundant.
- Identity integrity
  - Each node has an identity and can be found thanks to an identifier attribute.
  - Corresponds to Primary key constraint in relational databases.
- Referential integrity
  - Only existing entities can be referenced.
  - Each entity should have at least one relationship with an entity of another label to validate a Referential integrity constraint.
  - Corresponds to Foreign key constraint in relational databases.

Unlike the relational databases where the same structure must be maintained for each group of elements, the nodes and relationships in graph databases do not need to have the same number of attributes. If a value is not known or defined, we are not required to have this attribute. However, a default value can still be specified if necessary.
This is why the management of data in this type of database is quite complex and the addition of Integrity constraints can make it possible to solve this deficiency in NoSQL.

We will now look at the variants that can be found in other types of databases to learn more about data consistency management.

In a relational database context, with SQL for example, we can see that specific constraints are especially designed to manage data and its consistency.
In the resources [4] and [5], the subject of these constraints is addressed and the first one will focus in particular on *"Check"* constraints.

The first resource is actually a tutorial that will allow us to write *"Check"* constraints to validate the data in a set of columns.
The first important information about this tutorial is its definition of a *"Check"* constraint : *"It is an integrity constraint in SQL that allows you to specify that a value in a column or set of columns must satisfy a Boolean expression."*.

This resource also indicates the syntax of these constraints :

```
CONSTRAINT constraint_name CHECK(Boolean_expression)
```

In a relational case, the *"Check"* constraint will be satisfied when the Boolean expression returns the *True* value or the *Null* value.
In the following example, defined in the tutorial, we have a constraint that will ensure that the price of an item for sale is greater than zero.

```
CONSTRAINT valid_selling_price  CHECK (selling_price > 0)
```

These constraints are a simple way to verify data integrity in a relational context.

Then, the second resource about SQL is a global tutorial whose only part we will focus on is the part on constraints.
Another utility of the constraints proposed by this resource is to be able to limit the type of data that can be entered in a table. This will aim to ensure the reliability of the data.

The advantage of this tutorial is that it explains the different types of constraints that can be found in the relational context :

- Not Null constraint : It cannot have a *"Null"* value in the column.
- Default constraint : If no value is specified, a default value is assigned.
- Unique constraint : Within a column, all values must be different.
- Primary Key : Unique identifier of an element.
- Foreign Key : Reference the unique identifier of another table.
- Check constraint : This constraint is the one explained earlier.

Primary Key, Foreign Key and Unique Constraints are the ones that correspond to the Integrity constraints.

Afterwards, we can analyse an article [24] on a similar subject, also talking about *"Check"* constraints, but in a non-relational context this time.

The objective is to explain how to validate JSON values in a non-relational context on the basis of *"Check"* constraints.
The principle is exactly the same as in the relational context because they will specify that the value of Integers must be between 0 and 20 to assign a score to a TV show episode.

```
alter table scores
add check (rating >= 0)
add check (rating <= 20)
```

We can therefore conclude that *"Check"* constraints can also simply be implemented in a non-relational context.

Finally, we will look more closely at the non-relational context by focusing on resources addressing MongoDB [7] and Document-oriented databases [17].

The data within MongoDB is stored as documents thanks to a binary representation of JSON, called BSON.
In relation to data management, MongoDB offers two methods to connect documents :

1. Manual references : The purpose will be to save the identifier field of one document in another document as a reference.
2. DBRefs : These are references from one document to another using several fields, not only the identifier, to easily link documents together.

These two methods will make it possible to verify the links between the elements as a Referential integrity constraint could do.

As a conclusion to this section, we have seen that data integrity in the relational context is managed by *Primary Key, Foreign Key* and *Unique Constraints*. This therefore corresponds well to the Integrity constraints for the non-relational domain and, in addition, *"Check"* constraints could also be used to improve data consistency.

2.2.2 Referential Integrity Constraints

Now that we have an overview of Integrity constraints, we will focus on the specific type that will interest us for this work : Referential Integrity Constraint. A first approach of this subject has already been introduced in the section 1.3.

We will begin our analysis by discussing a first article [1]. A Referential integrity is a protection for the database which makes sure that the references between the data are valid and undamaged. In other words, we will focus on the validity of the relationships between the data.
Referential integrity can be compared to the RDBMSs[1] because they are each designed through the concept of *Foreign key* and *Primary key*.

These constraints allows several benefits :

- Improve data quality to preserve references.
- Make the development faster.
- Reduce the bugs amount thanks to better data consistency.
- Enhance the consistency through applications.

The syntax for representing Referential integrity constraint in a relational context, for example SQL, is as follows :

```
ALTER TABLE tableName1
ADD CONSTRAINT constraintName
FOREIGN KEY (columnList)
REFERENCES tableName2 [(columnList)] [onDelete] [onUpdate];
```

With the foreign key in *table1* and primary key in *table2*.

The notion of Referential integrity also applies to OO-DBMSs[2] through relationships between objects, that makes these objects dependent with each other. No matter the implementation platform that can be a RDBMS, an OO-DBMS, or even a programming language, it is required to deal with the Referential integrity.

In the case of a non-relational context such as NoSQL databases and more specifically Neo4J, we found a short article [11] which aims to show how the Referential constraints work in Neo4J.
In the case of a relational context, if a row of a table is deleted, Referential integrity constraints will ensure that the data related to this row will be impacted. This is more complex in NoSQL because if a node is deleted, it will also be necessary to delete all its relationships because a relationship must have a start and end node. As this article explains : *"Data integrity in Neo4j means that there should be no relationship without a start or end node and that there should be no property that is not attached to a node or relationship"*.
This is one of the issues that will be addressed in our work.

[1] Relational Database Management Systems

[2] Object-Oriented Database Management System

2.3 Technical background

All the tools and technologies required for a proper understanding of the continuation of the thesis will be described in this section.

2.3.1 Neo4J

Neo4J [13] is a NoSQL database management system which uses the Graph-oriented databases. It allows data to be represented in a graph as nodes connected by a set of arcs. It is mainly related to Cypher which is a language for querying the database.

Graph Database

As we said previously, Neo4J is based on Graph-oriented databases. These databases can apply CRUD[3] operations on a graph data model. The information is represented as nodes that have attributes and relationships to each other to indicate references.

We can pinpoint several advantages compared to other types of databases :

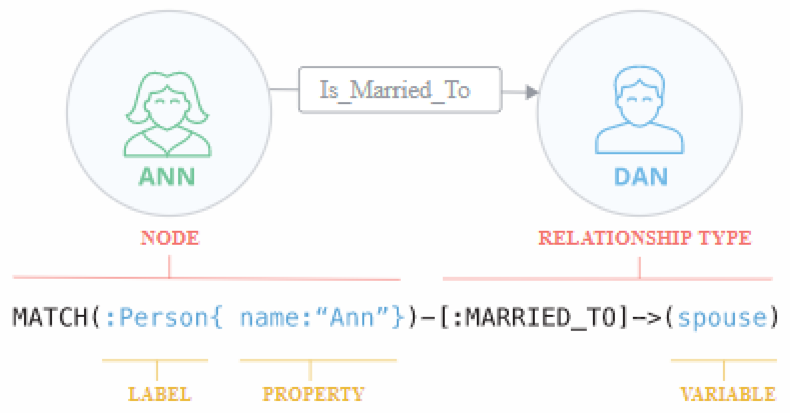
- Performance : Manage data growth over time.
- Flexibility : Modify the data structure to adapt to the evolution of applications without risk to current features.
- Agility : Easily adapts to today's agile and test-oriented development practices.

The Graph-database become very useful in situations where trees or network structures are found like for instance social graphs. It is no longer necessary to make joins between the primary keys, which simplifies the task.
More information about this type of NoSQL datastore is detailed in the article [14].

Cypher

Cypher is a language that allows to perform queries on the Neo4J database and it is composed of a set of clauses, keywords and expressions.
This language has the objective of doing operations on the dataset such as retrieving information about nodes or relationships, as well as modifying them. It also allows to create or delete nodes, relationships and information on the database.
As shown in the Figure 2.1, from the Neo4J website [13], we can use some information to find a specific node like its label or properties, corresponding respectively to its type and attributes, and its relationships.
In this example, we can translate the query by : *Give me each node which are of type "Person", which have the value "Ann" for the attribute "name" and who have a relationship of the type "MARRIED_TO" with a node "spouse".*

[3] Create, Read, Update and Delete



```
MATCH(:Person{ name:"Ann"})-[:MARRIED_TO]->(spouse)
```

Figure 2.1: Cypher representation

To build a Cypher query, several clauses are proposed :

- CREATE : Enables the creation of nodes and relationships.
- RETURN : Allows you to return a result. It can be a node, a relationship or simply a value of an attribute.
- MATCH : Allows you to select an element of the graph and then modify it or simply return its values.
- WHERE : Allows you to specify information in relation to an element.
- SET : Enables the modification of an element.

We will now show a simple example of how to use these clauses :

```
MATCH (node:Jurisdiction)-[relationship:Gender]->(nodeBis:Male),
WHERE node.id = 1000,
RETURN relationship.number
```

This query should return the number of “Male” persons who are linked to the Jurisdiction whose identifier is ’1000’.

Java Driver API 1.7

The Java Driver API aims to manage the Neo4J graph database directly from the Java code. This API is based on the Cypher language, described in the previous part. There are three steps to using this API : the database connection, the execution of queries and the disconnection from the database.

Connection

The connection is established as follows, with the URL of the Neo4J graph database as well as the identifier (user name) and the password.

```
Driver driver = GraphDatabase.driver(url,
                AuthTokens.basic(user, password));
Session session = driver.session();
```

Query

In the Java code, Cypher queries are created by a concatenation of String and executed by a *run* method.
The following code shows the execution of a query by the *run* method, with the possibility to provide additional data with the keyword *parameters*.
The result of a query will be stored in a *StatementResult* which can be reused later in the code.

```
String query = ‘‘MATCH node:Entity RETURN node”;
session.run(query);
```

Disconnection

After executing all queries, it is necessary to close the connection to the database. The disconnection is done by calling the following code :

```
driver.close();
```

JCypher

The JCypher API [12] provides another way to execute Cypher requests. It is considered mainly as a fluent Java API for Cypher whose objective is to simplify the creation of queries.
Its creator, *Wolfgang-Schuetzelhofer*, provided the GitHub repository of his project [25] with the source code and a *Wiki page* explaining how to use this API.

Connection

The connection is made in this way with JCypher.

```
Properties props = new Properties();
props.setProperty(DBProperties.SERVER_ROOT_URI, URL);
props.setProperty(DBProperties.DATABASE_DIR, ‘‘C:/NEO4J_DBS/01”);
dbAccess␣=␣DBAccessFactory.createDBAccess(DBType.IN_MEMORY,
props,␣user,␣password);
```

Query

This code shows the creation of queries with JCypher. It is no longer in the form of a unique String, but with a set of keywords.

```
JcQuery query = new JcQuery();
query.setClauses(new IClause[] {
CREATE.node(‘‘Name”).label(‘‘Entity”).property(‘‘Attribute”)
.value(‘‘Value”)
```

Disconnection

The disconnection with JCypher is made like this.

```
if (dbAccess != null) {
    dbAccess.close();
        dbAccess = null;
}
```

2.3.2 Neo4J example

In this section, we will explain an example of a small Neo4J project made in order to feel comfortable with the technology. The Java Driver API was mainly used in this exercise.

Context

The first step is to find a large dataset which will allow to perform queries on a large number of elements. For this purpose, we have looked after some *CSV* files that contain many interesting data in order to use them as a graph.

The file that we have selected for this example is about the rate of participation in an election. For each “Jurisdiction” of a specific country, the number and percentage of participants are mentioned based on two criteria. The first one is the *gender*, either male or female, and the other is the *community*, for example “Hispanic Latino” or “American Indian”. There are many different communities in this dataset but we only used two of them to have an easier graph to read and understand.

Creation of the dataset

The second step aims to set up the dataset by importing the file and creating all the nodes and relationships that we need for this.

File import

The following code will allow you to read the data from the file. We will have a *"While"* loop that will go through all the lines of the file and for each of them, we will be able to extract information and store them the *"jurisdiction"* array. We can then use this array to create the corresponding nodes.

```
String csvFile = ‘‘myFile.csv”;
BufferedReader␣br␣=␣null;
String␣line␣=␣‘‘”;
String cvsSplitBy = ”,”;

try {
    br = new BufferedReader(new FileReader(csvFile));
    while ((line = br.readLine()) != null) {
        String[] jurisdiction = line.split(cvsSplitBy);

        //creation of nodes and relationships here

    }
} // catch exceptions here
```

Creation of nodes and relationships

When these data were imported, it was necessary to create the nodes and relationships in the Neo4J database. First, we created the unique nodes, in other words the nodes corresponding to genders and communities. So we have a "Male" and a "Female" node for gender, as well as for communities with a "Hispanic Latino" and an "American Indian" node.

```
public void createGenderAndCommunity() {
  try (Session session = driver.session()) {

    //Gender
    String createNodeFemale = ”CREATE␣(node:Gender)␣” +
    ”SET␣node.message␣=␣'Female'␣” +
    ”RETURN␣node”;
    session.run(createNodeFemale);
    String createNodeMale = ”CREATE␣(node:Gender)␣” +
    ”SET␣node.message␣=␣'Male'␣” +
    ”RETURN␣node”;
    session.run(createNodeMale);

    //Community
    String createNodeHispLat = ”CREATE␣(node:Community)␣” +
    ”SET␣node.message␣=␣'HispanicLatino'␣” +
    ”RETURN␣node”;
    session.run(createNodeHispLat);
    String createNodeAmeInd = "CREATE␣(node:Community)␣" +
    "SET␣node.message␣=␣'AmericanIndian'␣" +
    "RETURN␣node";
    session.run(createNodeAmeInd);
  }
}
```

Then, we had to create the nodes corresponding to the “Jurisdiction”. Each line of the *CSV* file corresponds to a jurisdiction, so we will have a node for each line in the file, and the information contained within these lines will represent the information contained in the relationships with the “Gender” and “Community” nodes.

The following Java code represents the creation of a “Jurisdiction” node corresponding to a line of the *CSV* file and its relationships :

```
public void createNode(String name, int totalPerson,
int id) {

  try (Session session = driver.session()) {
    String createNode = "CREATE␣(node:Jurisdiction)␣" +
    "SET␣node.id␣=␣$id,␣node.message␣=␣$name,␣" +
    "node.totalPerson␣=␣$totalPerson␣" +
    "RETURN␣node";
    session.run(createNode, Values.parameters("name", name,
    "totalPerson", totalPerson, "id", id));
  }
}
```

Then, the code below is an example of the creation of a “Gender” relationship. The method for creating “Community” relationships will be built with a similar approach.

```
// Creation of a ‘‘Gender” relationship
public void createRelationshipGender(int number, double
percentage, String type, int id) {
  try (Session session = driver.session()) {

    // (number>0) verifies that there is at least one
    // corresponding person before creating the relationship
    if (number > 0) {
      String query = "MATCH␣(node:Jurisdiction␣{id:" + id +
      "}),␣(nodeGen:Gender␣{message:'" + type + "'})␣" +
      "CREATE␣(node)-[:Relation" + type + "{" + type + ":" +
      number + ",percentage␣:" + percentage + "}]->(nodeGen)";
```

```
            session.run(query);
        }
    }
}
```

Final graph

The graph is now built in Neo4J and it can be accessed through queries.

The Figure 2.2 shows the meaning of the colours of the graph, Figure 2.3, with the three types of nodes and the four types of relationships.

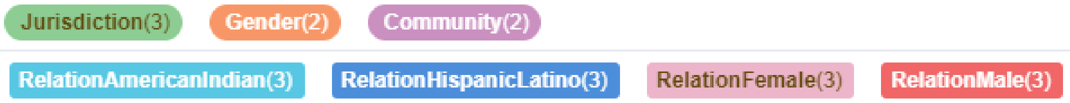


Figure 2.2: Neo4J example : Legend of the graph colors

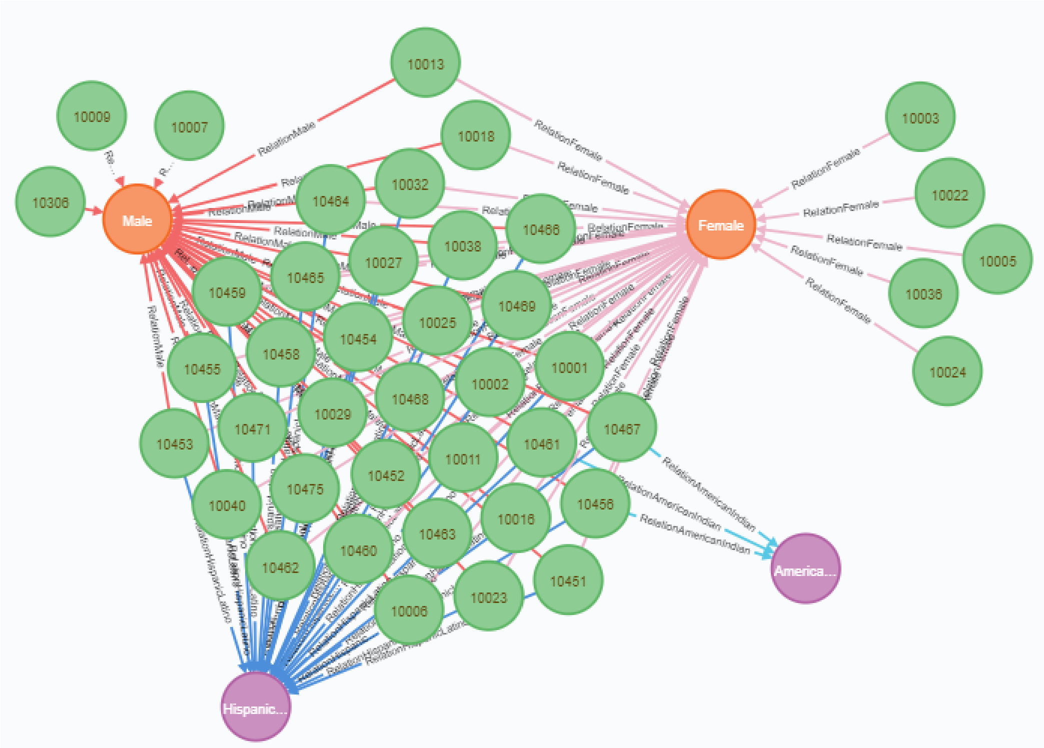


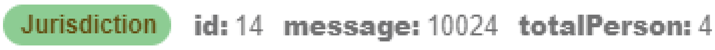
Figure 2.3: Neo4J example : Final graph

.

We can now see in the Figure 2.3 the graph representation of the first 40 lines of the *CSV* file. Each “Jurisdiction” node has a relationship with a “Gender” and “Community” node if and only if it has at least one person corresponding to this other node.

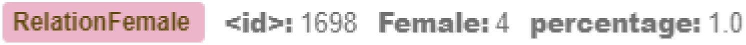
Eventually, information on jurisdictions is accessible through the attributes. A jurisdiction has attributes to represent its identifier (“id”), the total number of persons who participated in the election (“totalPerson”) and its name (“message”).

The attributes of a relationship contain information about a specific jurisdiction. For example, in the Figure 2.5, the relationship “RelationFemale” between a jurisdiction and the “Female” node of type “Gender” will give us the number of female person who participated in this specific jurisdiction and the percentage that this represents in relation to the total number of person for this specific jurisdiction.



Jurisdiction **id:** 14 **message:** 10024 **totalPerson:** 4

Figure 2.4: Neo4J example : Properties of a “Jurisdiction” node



RelationFemale **<id>:** 1698 **Female:** 4 **percentage:** 1.0

Figure 2.5: Neo4J example : Properties of a relationship

Queries

To conclude this example, several queries have been set up to access the data in this graph. The two APIs that will be needed in this work, Java Driver API and JCypher API, were used to understand how they work.

JCypher API

```
// Get information about ‘‘Jurisdiction” nodes
static void getHispanicLatino() {
  JcNode nodeJurisdiction = new JcNode("Jurisdiction");
  JcRelation rel = new JcRelation("rel");
  JcQuery query = new JcQuery();
  query.setClauses(new IClause[] {
    MATCH.node(nodeJurisdiction).label("Jurisdiction")
     .relation(rel).out().type("RelationHispanicLatino")
     .node(),
    WHERE.valueOf(rel.property("HispanicLatino"))
    .GT(3),
    RETURN.value(nodeJurisdiction).ORDER_BY("id"),
    RETURN.value(rel)
  });

  JcQueryResult result = dbAccess.execute(query);
  List<GrNode> listNode = result.resultOf(nodeJurisdiction);
  List<GrRelation> listRel = result.resultOf(rel);

  for (GrNode node : listNode) {

    //For each of the nodes, process the data.

  }
  return;
}
```

In this simple query with JCypher, the objective is to retrieve all the nodes of the label “Jurisdiction” that have at least four participants coming from the community “Hispanic Latino” in order to be able to use this data in our java code later.

Java Driver API

```
// Get information about relationship
public void getTotalPerson() {
  try (Session session = driver.session()) {
    String query = "MATCH␣(node:Jurisdiction)-" +
    "[relMale:RelationMale]-(nodeMale:Gender)," +
    "(node)-[relFemale:RelationFemale]-(nodeFemale:" +
    "Gender)␣RETURN␣node.message␣AS␣Message,␣" +
    "relMale.Male␣as␣Male,␣" +
    "relFemale.Female␣AS␣Female,␣relMale.Male␣+␣" +
    "relFemale.Female␣AS␣Total,␣relMale.percentage␣" +
    "AS␣MalePercentage,␣relFemale.percentage␣AS␣" +
    "FemalePercentage␣ORDER_BY␣Total";
    StatementResult result = session.run(query);
    while (result.hasNext()) {
      Record record = result.next();

      //For each of the nodes, process the data.
    }
  }
}
```

In this example, the “Match” clause selects the nodes that have a relationship with the node “Male” and the node “Female”. In other words, jurisdictions that have at least one male and one female person who participated in the election. The query will return some information about the two relationships like the number of male and female person and we can calculate with that the total number of person who participated to the election for a specific jurisdiction. We can also used an “Order by” clause to sort the values that the query will return.

2.3.3 Xtext for defining Domain-Specific Language

Xtext is the selected framework to provide the grammar definition language required for the Referential integrity constraints. When the grammar is defined, an Ecore metamodel is generated, as we will demonstrate in the Part 5.1. Using this technology will offer a grammar, an Ecore generated metamodel from the grammar and a model serializer to create examples based on the generated Ecore metamodel.

According to the Chapter 6 of the reference book [2], *“Modeling Languages at a Glance”*, a few DSLs principles have been set out :

- The language need to provide good abstractions to the developer, to be intuitive and make life easier, not more complicated.
- The language does not have to depend on an one-man expertise to adapt or to use it. Its definition has to be shared between the people and be accepted after some evaluation.
- The language has to evolve, and has to remain maintained following the context needs, otherwise it is sentenced to death.
- The language has to meet the supported methods and tools, because the domain experts maximise their productivity by working in their domain. And they do not want to spend time to create new methods or tools for the language.
- A good DSL has to respect the *open-close principle*, which means that software entities have to be open to extensions and closed to changes.

A few kinds of DSLs exists. The first one which is the *external DSLs* are created from scratch and there is needed to build a parser and an interpreter, the parser generator being ANTLR. A few advantages of this method are the freedom of notation, errors management and the support of static analysis and optimisation

Then the *internal DSLs* are represented in two categories :

- **Embedded DSLs** : DSL embedded into an existing functional or oriented host language, like Ruby or Scala.
  For instance, *RubyTL* is a Model-To-Model Transformation language embedded into Ruby, which is built as a DSL.

- **Fluent API** : An API is created following four techniques :
  - Method chaining : Invoke multiple method calls without needing to store the intermediate values.
  - Function sequence : Generate a list of sequential numbers.
  - Nested function : Function which is defined within another function.
  - Expression builder : Build expressions by selecting items in lists.

At last, there are the *DSL workbenches*, also known as "Metamodel-based tools". For this case, a parser, an injector, a generator and an editor are directly generated from a specification based on a metamodel.
It is firstly important to introduce the concepts of abstract and concrete syntax. The abstract syntax defines how the program looks like and the abstract syntax of some implementation is the tree representation. The concrete syntax is the set of rules that define how the program looks like. Two kinds of concrete syntaxes are supported by the existing frameworks : *Graphical Concrete Syntaxes* (GCS) and *Textual Concrete Syntaxes* (TCS). The focus will be on the TCS in this work. This kind of concrete syntax is used to work with textual documents and allows an user-friendly development for textual editors in modelling language. For the TCS, some of the most commonly used tools are *EMFText* and *Xtext*, whereas for the GCS, there would be *MetaEdit+* and *Sirius*.

2.3.4 Model-To-Text Transformation

As we saw in the introduction, the purpose of the Model-to-Text transformation will be to generate code.
The work made in the article [6] has similarities to this thesis. Indeed, the main contributions of the article are interesting for this work realisation.
The first of their stages will be to define a metamodel that allows to represent implicit structure in Graph databases. Then, they will do a mapping of the UML diagram to the Graph-oriented database. Finally, they will develop a framework that will generate the code to access the database.
Their developed tool *UMLtoGraphDB* has been made under these following steps :

1. Two inputs : A UML class diagram and OCL constraints.
2. The UML class diagram is transformed to have a GraphDB model and the OCL constraints are transformed to have a "Gremlin model".
3. The GraphDB model and the "Gremlin model" are going through the "Graph2Code transformation".
4. This transformation leads to the *Application Backend* and the Graph Database code is generated.

This article is interesting because it has a similar approach to ours, starting from the M2T technique to go to Graph-oriented databases.

To end this section, there are a few template-based transformation languages existing such as *XSLT*, *JET*, *Xpand*, *MOFScript* and *Acceleo*. This last will be used for the implementation (Chapter 5) and will be explained here.

Acceleo

Acceleo [8] will be the tool used in this work to perform the Model-to-Text transformation part. It is part of the Eclipse environment and is developed in Java.
Thanks to the slides of our host university [19] and [3], which teaches this tool, we were able to easily adapt to it.
Acceleo will need a model and metamodel at the input and will produce a file containing code at the output. Within the code, it will be necessary to create templates that will allow it to be structured.

Another tool that could have been used to perform this task is Xtend. In addition, it is part of the Xtext tool. Its syntax is very similar to Java, which makes it rather intuitive.
However, we have chosen to opt for Acceleo in this work.

Chapter 3

Methodology

This chapter will explain, in the first instance, the methodology to be followed by the end-user to apply several constraints to a database. We can divide it into two main steps : writing the Referential integrity constraints and applying these constraints to a database.
Then, we will look at the methodology used to develop our solution and the possible links with the first one.

3.1 Methodology end-user

As we said previously, this part will be devoted to the end-user methodology. The goal is to understand the path that the constraint will follow from the user's brain to the database. The figure 3.1 provides a schematic view of it.

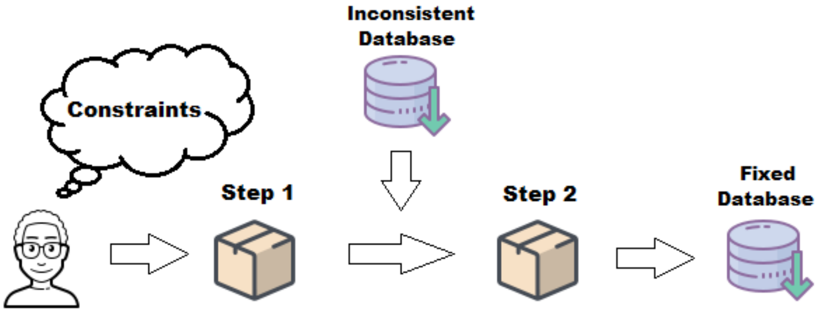


Figure 3.1: Diagram representing the steps of the methodology

In this schematic case, we have a person who wants to define several Referential integrity constraints on his database. The first step will be to write these constraints and for that it will be necessary to provide the user with a language allowing him to do that. The result of this first step will be all the RICs written and ready to be applied.

Then, we will use this result and the database targeted by the user to perform the second phase. The purpose of this step will be to apply the constraints to all the dataset. Finally, the result of this second step will be our corrected database.

3.1.1 Contextualization

We will now explain a basic and quick example to put this approach into context. We have a first entity *"nodeRed"* and a second *"nodeBlue"* which have relationships with each other.

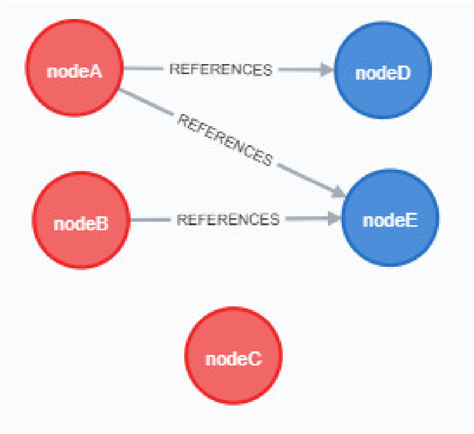


Figure 3.2: Initial data for the methodology example

The user can write Referential integrity constraints, corresponding to our Step 1. Let us take the case where he would like to no longer have a *"nodeRed"* without a reference in the database, that means a node who has no relationship with no other. The user will be able to write the following Referential integrity constraint : *For each "nodeRed" element, I want it to have at least one relationship with any of the "nodeBlue" elements.*
The written RICs will be stored in a file and sent to Step 2 which will apply them to the database.

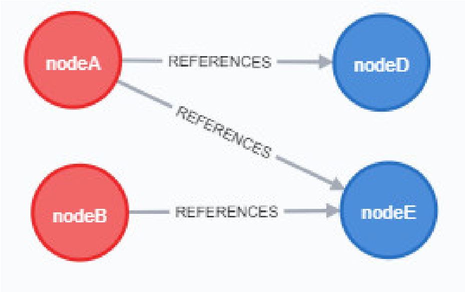


Figure 3.3: Final data for the methodology example

As we can see in the figure 3.3, the node of type *"nodeRed"* whose name was *"nodeC"* has been deleted from the dataset because it had no reference.
In the next chapter, **Design**, we will discuss the design of the Referential integrity constraints where we will explain in detail their structure and what can be done precisely with them.

3.2 Methodology of the solution

During our internship, we followed the approach of the end-user methodology to develop our solution by dividing our work into two main tasks, corresponding to those in the figure 3.1. A plan was established with the internship supervisor to distribute properly the work into several pieces and choose the most appropriate tools and technologies for each of them. In this chapter, we will also discuss about the different choices that have been made during this work.

3.2.1 Step 0 : Draft of the RIC metamodel

The initial step in apprehending the domain was to define a draft of the RIC metamodel, with the *EMFText* tool. This technology, where EMF means *Eclipse Modeling Framework*, is a tool to define the textual syntax of Ecore-based metamodels.

The following meta-model, Figure 3.4, was then created to represent the design of our first idea of what a Referential integrity constraint would be.

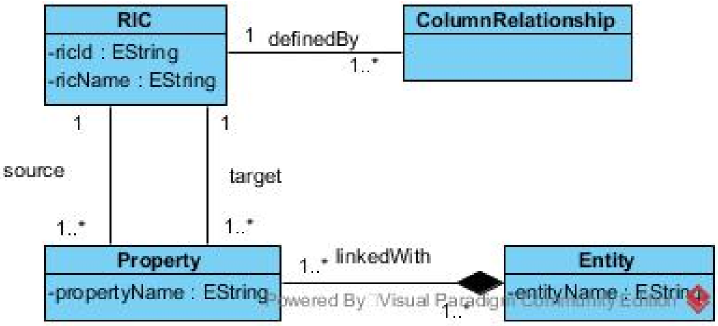


Figure 3.4: Initial metamodel of the RIC-DSL

As we can see on this metamodel, a Referential integrity constraint is identified by its **ricId** and is defined by one or many **ColumnRelationship**.

The main concept in this diagram is the relationships **Source** and **Target** between the RIC and **Property** classes. It means that a RIC can have multiple properties "Source" and multiple properties "Target". At last, these properties are linked with at least one **Entity**.

This metamodel was the starting point of our project and the first version of our structure of a RIC. During the progress of our work, we have made numerous changes to this structure by adding the notions of cardinality, bi-directionality, action and condition. We will discuss this more technical subject in the next chapter, **Design**, where we will present the evolution of this metamodel until we get its final version.

3.2.2 Step 1 : Writing Referential integrity constraints

This step will aim to propose to our user a way to write his constraints. Based on the final meta-model of a RIC, we will have to define a language, simple and intuitive, that will meet the needs of the end-user.

To achieve this, we must define a concrete syntax to write RICs based on the *XText* tool. This task consisted of four sub-steps :

1. First of all, a learning of the Xtext framework had to be done to master the tool.
2. A simple DSL was then created with the concrete syntax defined and an editor to write RICs has been developed.
3. Afterwards, the capabilities of XText like the coloring syntax and the content assist were explored to implement a more advanced and customized editor.
4. To finish this task, several tests based on the previous DSL were implemented for different datasets.

3.2.3 Step 2 : Apply RICs to check database

The second step will aim to apply the RICs written by the user on a database. We will have to use the Model-To-Text transformation techniques to achieve this and in particular the *Acceleo* tool, to obtain the information from the model provided.

This task has taken place in five different sub-steps :

1. The first step was to learn how Acceleo was working.
2. To get used with the Graph-oriented databases, a study of the *Neo4j Java Driver* and *JCypher* APIs were required.
   The concrete example in the section 2.3.2 has been realised to show how to use Neo4J in our context.
3. After that, we have started to develop the code. We created several methods to verify the validity of a RIC according to its type and its characteristics.
4. When these methods were defined and tested, it was time to create the Model-To-Text transformation with Acceleo to obtain the constraints written by the user.
5. At last, all that remained was to generate the due code representing the RICs and execute it to fix the database.

Chapter 4

Design

Starting from the initial meta-model of a Referential integrity constraint, Figure 3.4 in the previous chapter, a few additions have been made through the time thanks to some discussions with different members of the research group and during the visit of our supervisor. This chapter will explain how the vision of a generic Referential integrity constraint has evolved in our minds, to lead to the final structure of a RIC which is used to control the consistency of the data in a database.

4.1 Structure of a relationship

As the design progresses, we have determined that a RIC can be of two main types : Classic relationship or Tag relationship. These are the following :

4.1.1 Classic relationship

The classic relationship is a Referential integrity constraint based on attributes and is defined in the following way :

```
entity1.attribute1 -> entity2.attribute2
```

The goal will be to compare the values of the attribute of a first entity with the values of the attribute of a second entity.
To have a valid relationship, every value contained in *attribute1*, belonging to *entity1* has to match to a value of *attribute2*, belonging to *entity2*.

4.1.2 Tag relationship

The second kind of relationship is the reference between two nodes with an arc whose label will be the **Tag**. This type of relationship is defined like this :

```
entity1 -(tag)-> entity2
```

Every node belonging to *entity1* should have at least one arc, with the *Tag* name, with an *entity2* element.

4.1.3 Adding bi-directionality

Afterwards we can add two new types of relationships for an RIC, based on the previous ones.
We chose to add the fact that a relationship can be **bi-directional** which would mean that the previous definition will be for both ends of the relationship arrow. The relationship will first be analyzed from left to right and then from right to left, as if they were two simple relationships.

Thus, the relationships will be represented like this, going from the Classic relationship to the Tag relationship, respectively :

```
entity1.attribute1 <-> entity2.attribute2

entity1 <-(tag)-> entity2
```

4.2 Cardinalities

The cardinality defined the minimal and maximal number of possibility that an object has in a relationship. It will be necessary to verify if the value is in this interval.

In an unidirectional relationship, a cardinality can be added to the destination part of the arrow (for example after *entity2*). While in bi-directional relationships, cardinalities can be added to both parts of the relationship at the same time. Putting cardinalities is not mandatory but it can be interesting for consistency reasons.

Here is an example of their representation :

```
entity1.attribute1 -> entity2.attribute2[MIN..MAX]
entity1 -(tag)-> entity2[MIN..MAX]

entity1.attribute1[MIN..MAX] <-> entity2.attribute2[MIN..MAX]
entity1[MIN..MAX] <-(tag)-> entity2[MIN..MAX]
```

While *MIN* is the minimal cardinality and *MAX* the maximal one. Here follows the four different couples of cardinalities that the user can face in his RIC declaration.

- [0..1] : It means that there should be at most one link between the two parts of the relationship.
- [0..N] : There will be an undefined number of links for the relationship.
- [1..1] : In this case, we exactly have one relationship.
- [1..N] : There is at least 1 link between both parts of the relationship.

4.3 Actions

Actions are defined in order to improve the consistency of the datasource. The actions are not related to the validity of a RIC, that means that they will be performed regardless of whether the constraint is valid or not and it will allow the user to adapt the behaviour of the database. We decided to represent them under four different cases, that will be explained in this part.
It is important to indicate that this feature will be optional for the user but it will be his only way to modify the datastore.

4.3.1 Adding information

This first action will add some information in the dataset and its behaviour depends on the relationship type. In the case of a Classic relationship, an edge will be created, if it does not already exist, between the node of *entity1* and a node of *entity2* whose its *attribute2* corresponds to a value in the *attribute1*. In other words, every value of the *attribute1* will be taking in account, looking if there is an existing edge with the *entity2* node whose its *attribute2* corresponds to this value. If the edge does not exist, it will be created following the information put by the user. This step is done for each node of type *entity1*.

We will illustrate this action with the Figure 4.1.

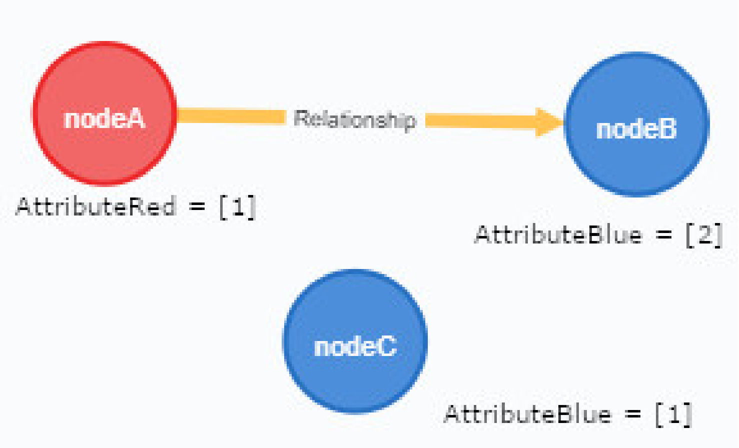


Figure 4.1: Action Add Info : Before

In the case of a Classic relationship :

```
entityRed.attributeRed -> entityBlue.attributeBlue
```

With the action “Add Info”, the program will add an edge between *nodeA* and *nodeC*. The value of *attributeRed* corresponds to the value of *attributeBlue* of *nodeC* meaning that there is a reference between *nodeA* and *nodeC*, but there is no arc on the graph. The “Add Info” action will allow you to add this one.

For a Tag relationship, a checking is made to know if there are relationships between *nodeRed* nodes and *nodeBlue* nodes, and if there are some, the identifier of the *nodeBlue* node is added in the attribute of the *nodeRed* node.
If we take again the Figure 4.1, in the case of a Tag relationship :

```
entityRed −(Tag)−> entityBlue
```

The “Add info” action will add the value of the identifier of *nodeB* in the *AttributeRed* of *nodeA* because we have an edge between *nodeA* and *nodeB*, meaning that there is a reference between them.

We can see, on the figure 4.2, in **red** the changes made by these two actions : an arc has been added and a value has been added to *nodeA*.

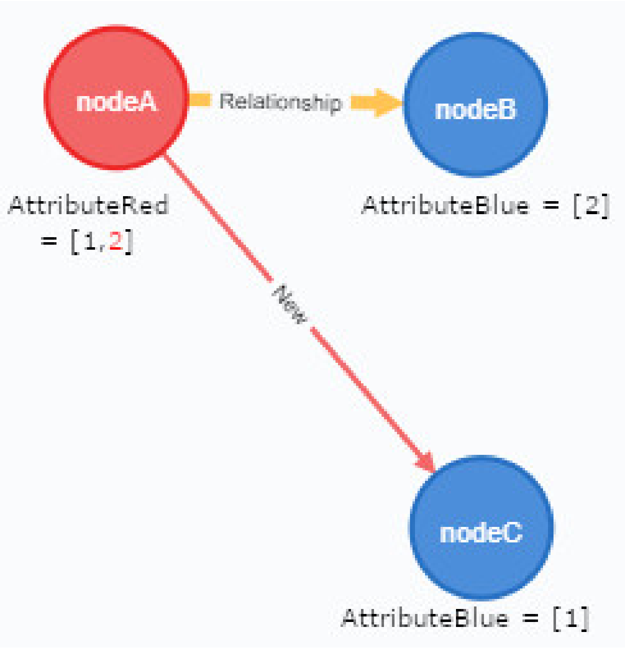


Figure 4.2: Action Add Info : After

The goal of this action is, for each node of the entity *nodeRed*, to have consistency between its attribute *AttributeRed*, referencing the *AttributeBlue* of the entity *nodeBlue*, and its relationships, targeting node of entity *nodeBlue* as we can see in the Figure 4.2. In conclusion, the consistency of the data is both the reference by arcs and by attributes.

To be as generic as possible and to only not be close-minded to graph-oriented database, the choice was made to call this action “Adding information” instead of “Adding edge” or something else only related to this specific kind of non-relational datastore.

4.3.2 Deleting information

The purpose of this action will be to delete an information or node. Caution should be exercised with this type of action, as it is always risky to delete data from the database.

In the case of Classic relationship, each value of the *AttributeRed* are taken. We will check if there is an *nodeBlue* that have this value in its *AttributeBlue*. If no match exists, the value of the *AttributeRed* is removed.

For Tag relationships, a node of the source entity *nodeRed* should have at least one relationship with a node of the target entity *nodeBlue*. If there is no relationship, the first node is deleted.

4.3.3 Deleting information in cascade

Another kind of deletion of information has been found out. Even though in the case of Classical relationship it will not change anything, it will have an interest for Tag relationships. If a source node is removed, and this one has other relations with other entities, this action will permit to delete all of his relative nodes in cascade.
You have to be very careful with this kind of action that can remove many elements from the database.

4.3.4 Showing information

The idea here is just to show some information about a node like its attributes, no matter the kind of relationship. Indeed, it will return all the information about the node and entity, and its direct relationships with other entities.

4.3.5 Summary

This section will provide a summary table of each action, with its respective description, the related section in the thesis structure and if there is an update in the database.

Action Name	**Description**	**Related Section**	**Update in the database**
Add Info	Adding information in the dataset	Section 4.3.1	Yes
Delete	Deleting information depending on the kind of relationship (classic/tag)	Section 4.3.2	Yes
Delete Cascade	Deleting all the relative nodes of the chosen entity in tag relationships	Section 4.3.3	Yes
Info	Showing the information about an entity	Section 4.3.4	No

Figure 4.3: Summary table of the Actions

4.4 Condition

The concept of condition here is to act as an extra-validation for the Referential integrity constraint by check if the attribute value in the dataset complies with the written condition to validate the RIC.

```
Entity.attribute {RelationalOperator} value {LogicalOperator}
```

The initial goal is to compare a property with a value using one of these five different relational operators : equals (=), greater than (>), lesser than (<), greater than or equals (≥), lesser than or equals (≤).

A property is the value of an attribute of a particular entity represented like this : **entity.attribute** . The compared value can be either a String, an Integer or a Boolean. This condition can be linked to one or a few more thanks to these both logical operators that are "AND" and "OR". In this case, it will be directly related to the following condition.

Here is an example to clarify this concept of condition with the help of the concrete example of the section 2.3.2 on elections within Jurisdictions.

```
(Jurisdiction.numberMale > 0) AND (Jurisdiction.numberFemale > 0)
```

This means that each jurisdiction must have at least one male and one female person who participated in the election. If both parts are good, it means that the condition is respected so the RIC can be validated.

4.5 Final metamodel

The figure 4.4 represents the final state of our Referential integrity constraint after integrating all the elements discussed in this chapter.
All classes correspond to the elements of this chapter, except the *"RicDSL"* class which is new and represents a list including all RICs.

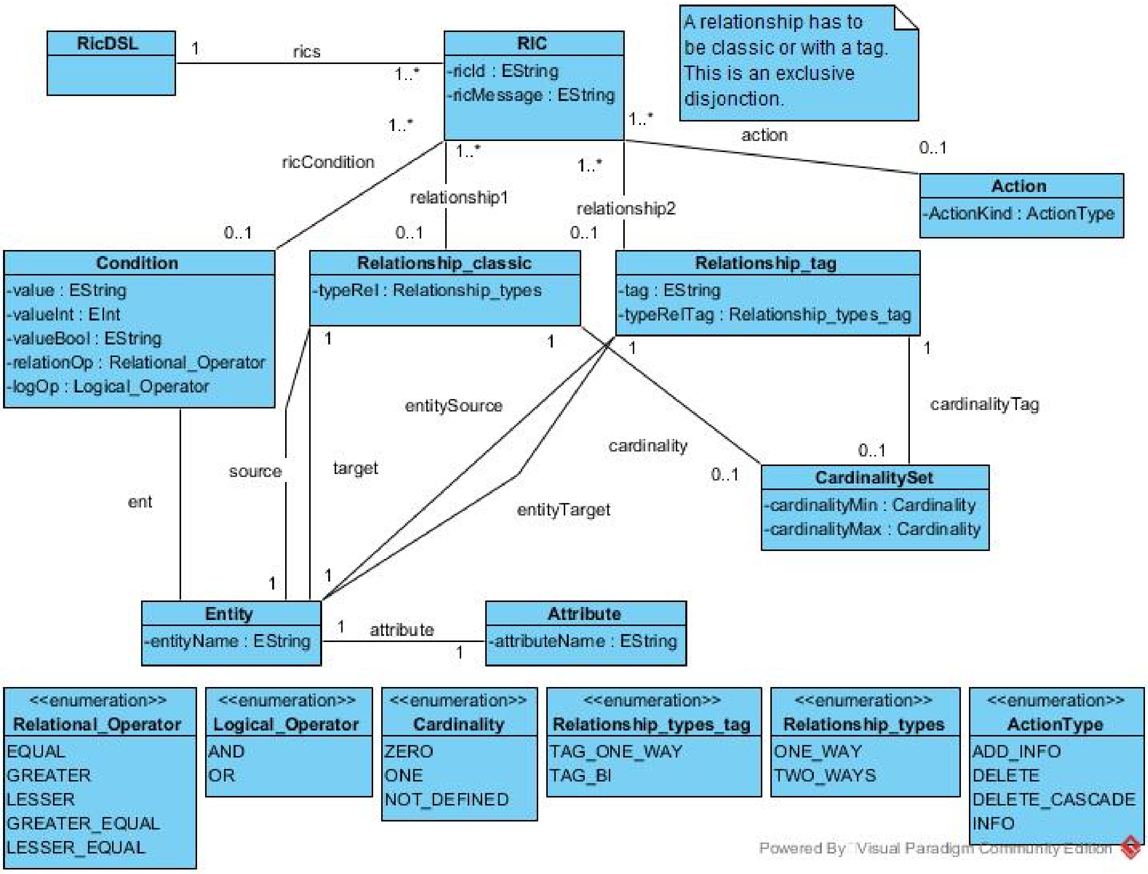


Figure 4.4: Metamodel of a RIC after the Design step

Chapter 5

Implementation

To create the Referential integrity constraints definition system on NoSQL datastores, two main steps were defined according to the Methodology chapter.
The first one was to create a Domain Specific Language (DSL) where the grammar of RICs definitions is created and defined to have a referring metamodel (*.ecore* file) to this grammar. This step was made through the XText framework, which is explained in the Part 2.3.3.
The second step is intended to develop the Java code to validate RICs. To do this, it is necessary to build the code to send requests to the Neo4J datastore. When the Java code is functional, a Model-To-Text transformation with Acceleo can be realised to generate the targeted code.

The implementation has been done on the Eclipse IDE in order to match the tools used and the code is available on the following GitHub repository : *https://github.com/fjavierbr/belgian-projects* .

Project hierarchy

The project is composed of five packages :

- org.xtext.stage.ricdsl
- org.xtext.stage.ricdsl.ui
- org.xtext.stage.ricdsl.ide
- org.xtext.stage.ricdsl.tests
- org.xtext.stage.ricdsl.ui.tests

The package *org.xtext.stage.ricdsl* contains three sub-packages : *src*, *src-gen* and *model*. The grammar of the **ricdsl** language, corresponding to the file *RicDsl.xtext*, is in *src*, while all the generated code is in *src-gen* and the .ecore generated metamodel will be in the sub-package *model*.
The second package *org.xtext.stage.ricdsl.ui* contains all the code related to the advanced features described in the Section 5.1.2. All the files written will be described in detail in this Section.
The last three packages are not relevant for the rest of this work.

5.1 Defining a DSL representing the grammar

The DSL has been defined following the **Design** chapter (Chapter 4).

5.1.1 Ricdsl language definition

Here is how the DSL has been written within the Xtext framework. The idea is to show the grammar code and then to explain it, with references to the Design chapter to point out how it is represented in the language definition.

Some Xtext features used in the language definition have to be expressed before showing it :

- Defining a keyword by writing a String between two simple quotes like this : ‘Keyword’.
- The symbol “+=” (example : attribute+=Entity) means that an *Entity* is added to the feature *attribute*.
- There are four possible cardinalities : exactly one is the one by default, zero or one is represented by a ‘ ? ’, zero or more by ‘ * ’ and one or more is represented by ‘ + ’.

RicDSL entity

```
RicDSL:
        (datasource+=Datasource) '\n\n'
        (rics+=RIC)*
        ;
```

The RicDSL entity may be compared to the root of the generated metamodel of a RIC. It contains the Datasource entity which will explained right below and a or more than one RIC. The ‘\n\n’ token is to go two lines lower after all the information of the Datasource entity has been given.

Datasource

```
Datasource :
        'DataSource' '(url='(url=ID) ',usr='(username=ID)
        ',pwd=' (password=ID) ')'
;
```

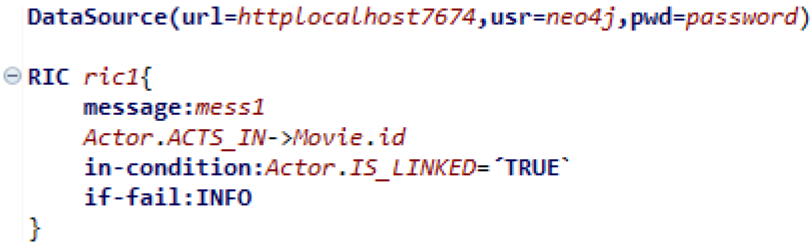
The keyword *DataSource* is provided to the user in the editor and directly after that, the related URL of the datastore is asked, following with the user and then the password required to access to the database, like it is in the Neo4j Desktop application. One and only one String will have to be enter by the user, respectively for the url field, the username and the password.

RIC

```
RIC :
        'RIC' ricId=ID '{' '\n\t'
        'message:' ricMessage=ID'\n\t'
        ((relationship1+=Relationship_classic) |
        (relationship2+=Relationship_tag)) '\n\t'
        'in-condition:' (ricCondition+=Condition)*'\n\t'
        'action' (action+=Action)? '\n'
        '}' '\n'
        ;
```

The keyword *RIC* is followed by the id to give to this RIC, then by a '{' to signal the beginning of the RIC, in the image of this symbol in Java, for instance. The ricId is mandatory to write and there can only have one. The '\n\t' notation means to go to the next line and make a tabulation space.
Then comes the keyword *message* followed by a message that the user can write. The next line is about the different kinds of relationship that will be explained below, according to what has been explained in the Chapter 4 *Design*. The '|' symbol is here to represent the OR operator which means that the relationship has to be a Classic or a Tag relationship, but not both.
Afterwards comes the keyword *in-condition* which will be followed by one or several conditions according to the section 4.4.
The last information to give in a RIC by the user is the action (Section 4.3). This part is not mandatory, the user can leave this blank or can enter an action. Then follow the symbol '}' to end the RIC.

The figure 5.1 represent how a RIC is written in the editor.



```
DataSource(url=httplocalhost7674,usr=neo4j,pwd=password)

RIC ric1{
    message:mess1
    Actor.ACTS_IN->Movie.id
    in-condition:Actor.IS_LINKED=`TRUE`
    if-fail:INFO
}
```

Figure 5.1: Representation of a RIC in the editor

Relationships

```
Relationship_types:
        ONE_WAY = '->' | TWO_WAYS = '<->'
;

Relationship_types_tag:
        TAG_ONE_WAY = '-('tag=ID')->'| TAG_BI = '<-('tag=ID')->'
;
```

```
Relationship_classic:
        (source+=Entity) (cardinalityLeft+=CardinalitySet)?
        (relationship+=Relationship_types) (target+=Entity)
        (cardinalityRight+=CardinalitySet)?
;

Relationship_tag:
        (entitySource+=Entity) (cardinalityLeft+=CardinalitySet)?
        (relationship+=Relationship_types_tag)
        (entityTarget+=Entity) (cardinalityRight+=CardinalitySet)?
;
```

As seen in the section 4.1.1 and the section 4.1.2, there are two kinds of relationship. These kinds of relationship can also be bi-directional, this is what the second token is in the *Relationship_types* and *Relationship_types_tag* entities, while the left part represent the uni-directional relationship. A relationship contains the entity at the left of the arrow, called here *source* or *entitySource* of the *Entity* type (see below). Such as the entity at the right part of the arrow (*target* and *entityTarget*), they may have a cardinality, which is not mandatory. The implementation about the cardinality will be explained a bit further. The figure 5.2 shows a Tag relationship that is written in the editor, while the figure 5.1 represents a Classic relationship.

```
Actor-(ACTS_IN)->Movie
```

Figure 5.2: Representation of a tag relationship in the editor

Entity

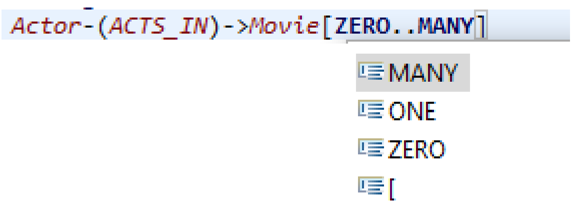
```
Entity :
        entityName=ID('.'(attribute+=Attribute))?
;

Attribute :
        attributeName=ID
;
```

An entity has its name and may be related to an attribute, like it is described in the section 4.1. It depends on the relationship type that is faced.

Cardinality

```
CardinalitySet:
          ’[’ (cardinalityMin+=Cardinality) ’..’
          (cardinalityMax+=Cardinality)’]’
;

Cardinality :
          ZERO = ’ZERO’ | ONE = ’ONE’ | NOT_DEFINED = ’MANY’
;
```

As described in the section 4.2, the minimal and the maximal cardinality of an attribute or an entity has to be represented among four possibilities. Here is to what the user is face when he wants to add cardinalities.



```
Actor-(ACTS_IN)->Movie[ZERO..MANY]
```

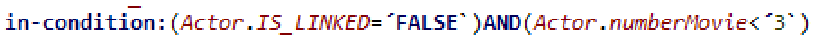
Figure 5.3: Representation of the cardinalities in the editor

Condition

```
Condition:
          (’(’)?(ent+=Entity) (relationOp+=Relational_Operator)
          ’  ’ ((value=ID)|(valueInt=INT)|(valueBool=(”TRUE”|
          ”FALSE”)))’‘’(’)’)? (logOp+=Logical_Operator)?
;

Relational_Operator:
          EQUAL = ’=’ | GREATER = ’>’ | LESSER = ’<’ |
          GREATER_EQUAL = ’<=’ | LESSER_EQUAL = ’>=’
;

Logical_Operator:
          AND = ’AND’ | OR = ’OR’
;
```

Seeing the section 4.4, the relational operators are represented in *Relational_Operator*, while the logical operators correspond to the *Logical_Operator* entity.
Each operator is represented by its symbol, like explained in the related part.

A bracket can be put by the user, but it is not mandatory. Its use is only to make the condition most easy to read for the user if this one will be long. The value that will be compared by a relational operator has to be either a String, an Integer or a Boolean. The figure 5.4 is an example about how a condition can be written in the editor.



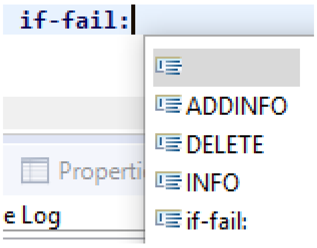
```
in-condition:(Actor.IS_LINKED=´FALSE`)AND(Actor.numberMovie<´3`)
```

Figure 5.4: Representation of a condition in the editor

Action

```
Action:
        ADD_INFO  =  'ADDINFO'  |  DELETE  =  'DELETE'  |  INFO  =  'INFO'
;
```

Following what has been explained in the section 4.3, three actions are available to choose by the user. It was not possible for a timing issue to implement the "Delete Cascade" semantic part and this is why it is not in the grammar. We will provide details about that issue in the Chapter 9. The figure 5.5 shows what the user can face in the editor when he wants to add an action to a RIC.



```
if-fail:
```

Figure 5.5: Representation of actions in the editor

This ends the Xtext grammar code that will generate an .ecore metamodel. After that comes the serialization where a dynamic instance of the metamodel will be created on the "RicDsl" entity. A *.xmi* file containing all the relevant data concerning every entities and attributes of the metamodel will be generated. A relevant example of a serialization is in the Appendix B.1. This file will be the input for the further step regarding the code generation, that will be explained in the section 5.2.

5.1.2 Advanced features

According to the **Xtext - Eclipse Support** [23], a few additional features for the user interface of the created editor can be explored. These are features such as : quick fixing, syntax coloring, outline view, hyperlinking or content assist. So, it became interesting to explore some of these advanced features. And the explored ones for this work are the **Syntax coloring** and the **Content assist**.

Syntax coloring

First of all, here is how the **RicDslUiModule.xtend** file has to be written to accept the new syntax coloring. It is important to point out that Xtext provide a default syntax coloring. The motivation was to have something more customised than it was.

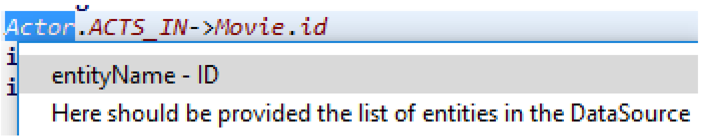
```
class RicDslUiModule extends AbstractRicDslUiModule {
        def Class<? extends IHighlightingConfiguration>
        bindISemanticHighlightingCalculator () {
           RicDslHighlightingConfiguration
    }
}
```

Then, the file **RicDslHighlightingConfiguration.xtend** had to be written, implementing **IHighlightingConfiguration** to override the *configure* method and then being able to update the syntax coloring and font of the keywords and the plaintext to make it more personal. The linked code to that is in Appendix A.1 and is quite easy to understand.

Content assist

The related code to this part is in the Appendix A.2 where three methods have been overrided :

1. *complete_Entity* where you have to write an Entity in the editor, you have the String "Here should be provided the list of entities in the DataSource." For timing issue, this String was shown instead of the list of the entities coming from the *Datasource*. The Section 9.2 explains how it should look like in the future.



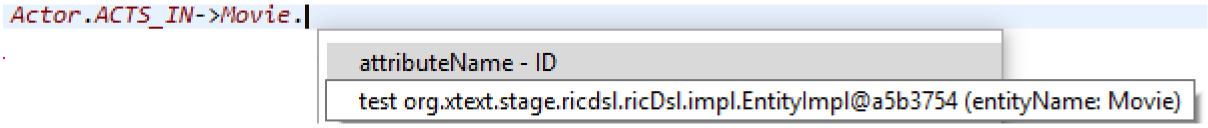
Actor.ACTS_IN->Movie.id

entityName - ID

Here should be provided the list of entities in the DataSource

Figure 5.6: Content assistant for an Entity

2. *complete_Attribute* where the goal here is to give the information about the *Entity* where the attribute to enter belong. The method call *currentModel* give the information about what is the name of the Entity as it can be read on the figure 5.7. The content assist shows the name of the Entity where the attribute belongs.



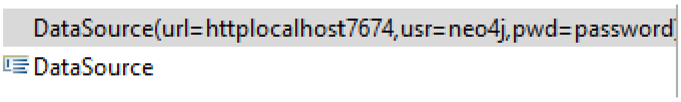
```
Actor.ACTS_IN->Movie.
```

attributeName - ID

test org.xtext.stage.ricdsl.ricDsl.impl.EntityImpl@a5b3754 (entityName: Movie)

Figure 5.7: Content assistant for an Attribute

3. *complete_Datasource* where the default information for the connection to a Neo4j database is given, like shown in the figure 5.8. The content assistant will give this information when you enter the *DataSource* keyword.



DataSource(url=httplocalhost7674,usr=neo4j,pwd=password

DataSource

Figure 5.8: Content assistant for the DataSource

5.2 Model-To-Text transformation

5.2.1 Introduction

This section will explain the implementation of the code allowing the validation of RICs, as well as the choices that result from it. It will be composed of two important parts: the code to send queries to the NoSQL datastore and then the code to perform the Model-to-Text transformation with Acceleo.

Project hierarchy

The project is composed of two important packages :

- org.eclipse.acceleo.dsl.uml.gen.java.main
- org.eclipse.acceleo.dsl.uml.gen.java.files

The first one, *main*, contains the Acceleo files corresponding to the templates that will generate the target code. There are two files : *neo4JApiJava.mtl* corresponding to the Java Driver API version and *neo4JApiJCypher.mtl* corresponding to the JCypher version.
The second package, *files*, is the place where the files resulting from the Model-to-Text transformation will be generated. *TransformationJavaApi.java* is the file generated by Acceleo for the Java Driver API version and *TransformationJCypher.java* is the file generated for the JCypher version.
Another important file in this package will be the one where the results of the RICs will be stored after being tested : *RICResult.json*. This file will save the results of the RICs in order to be able to consult them or to use them again later.

Starting hypothesis

Before starting the implementation of this part, we made an assumption about the structure of the nodes of the Neo4J datastore : Each node in the dataset is identified by a property *"id"* that is an Integer.
It is much easier for us to manage nodes in this way. This attribute will be unique and will allow us to browse through all the nodes in a simple and efficient way. This can obviously be modified and adapted later according to future needs.

5.2.2 JCypher and Java Driver API

As indicated several times since the beginning of this thesis, we will use two different APIs to access the Neo4J database and therefore have two identical versions of the code for each of them.
The goal will be to be able to compare these two APIs and we will analyze the creation of queries with each of them in the next chapter, **Taxonomy of Queries**.
In the rest of this chapter, we will focus the Java code independently of the two APIs. This code will therefore be identical in both versions because it does not directly contain queries to the database.

5.2.3 Develop the Java code corresponding to Neo4J

For this part, we will not use Acceleo yet. The goal is to programmatically define how to validate RICs using Neo4J. It will provide us with the basic models for code validation. We will use handwritten RICs in the code to perform our tests until we get to Acceleo. Once this is functional, we will integrate this code into the Acceleo templates, which is the next step in the work.
The *"validateRICJavaDriver"* file in the project is the one corresponding to this part of the work. The focus will be on queries to access the Neo4J datastore.
We wanted to have a *"main"* method as simple as possible, with only one processing method, called *"validateModel"* in which the RICs processing will take place, as well as the two connection/disconnection methods to the database (Section 2.3.1).

Methods for validating RICs

Our goal will be to develop 8 methods covering the possible types of RICs, we will then add the actions and conditions that are independent methods.
As we said, eight methods are enough to cover all cases of validation :

- Classic relationship without cardinality
- Classic relationship with cardinality
- Classic relationship without cardinality and bidirectional
- Classic relationship with cardinality and bidirectional
- Tag relationship without cardinality
- Tag relationship with cardinality
- Tag relationship without cardinality and bidirectional
- Tag relationship with cardinality and bidirectional

Explaining each of these methods would be highly repetitive and tedious. That is why we have decided to present only two of them, *Classic relationship* and *Tag relationship* without cardinality, and to briefly explain the others and their differences afterwards.

Classic relationship without cardinality :

We will start with the simplest method, corresponding to *Classic relationship*, whose purpose will be to verify that all the values present in the *attribute1* of the first node correspond to a value existing in the *attribute2* of one of the nodes of the second entity.

```
for(int id : nodeFirstEntity) {
        numberElements = getValue(id, firstEntity,
        firstAttribute);
        for (int i = 0; i < numberElements.size(); i++) {
                Object value = numberElements.get(i);
                if (!resultSecondQuery.contains(value)) {
                        if (!idNodeFail.contains(id)) {
                                idNodeFail.add(id);
                        }
                }
        }
}
```

We will analyse the corresponding code above.
For each node of the first entity, whose identifiers will be kept in list *"nodeFirstEntity"*, we will check that the values of its attribute, accessible through the list *"numberElements"*, correspond to a value of a node of the second entity. The *"resultSecondQuery"* list represents all the values of the specified attribute of the nodes of the second entity.
Nodes that do not meet this condition will have their identifier added to the list *"idNodeFail"*.

For the rest of this method, in the code below, we will need this list of failed nodes. In case this list is not empty, we know that at least one node has failed, so the RIC will not be validated and we do not need to test the condition, even if there is one.
If this list is empty, it means that each node is valid, we can then test the condition if there is any.

"listCondition" is a list containing all subconditions. If it is empty, so if there is no condition, we can return the result of the RIC. In the opposite case, we will have to test the condition(s).
If the global condition is valid, the RIC will be validated, otherwise it will fail.

```
if (!idNodeFail.isEmpty()) {
  validRIC = false;
  for (int id : idNodeFail) {
    LinkedHashMap<String, Object> attributeNode =
    infoNode(firstEntity, id);
    nodeFail.add(attributeNode);
  }
} else {

  if (!listCondition.isEmpty()) {
    condition = checkCondition(listCondition);
    for(Entry<String, Boolean> entry :
    condition.entrySet()) {
      String key = entry.getKey();
      if (key.equals("result")) {
        validCondition = entry.getValue();
        if (validCondition) {
          validRIC = true;
        } else {
          validRIC = false;
        }
        break;
      }
    }
  } else {
    validRIC = true;
  }
}
```

The method is not yet complete, we must now test the actions. If there is one, we will call the method *"actionAttribute"* corresponding to the actions for Classic relationships.
The methods corresponding to actions and conditions will be presented later in this chapter.

```
if (!action.equals("")) {
  resultAction = actionAttribute(action, firstEntity,
  secondEntity, firstAttribute, nodeFirstEntity,
  idNodeFail);
}
```

Finally, the code below ends our method.
The *"generateResult"* method will allow to put all the useful information about the RIC in a list, *"listResultRIC"*, which we will return.

```
listResultRIC = generateResult(name, validRIC,
nodeFail, action, resultAction, condition);

return listResultRIC;
```

Tag relationship without cardinality :

Now will be discussed the second type of relationship. For *Tag relationships*, the purpose is to verify that there is at least one relationship between a specific node of the first entity and any node of the second entity.
"listIdEntity" corresponds to the list of identifiers of all nodes of the first entity. We will therefore make a query, for each identifier, that will return the relationships of this node with a node of the second entity.
If the result of the query is null, then the RIC is not valid and the identifier is added to list *"idNodeFail"* which corresponds to the identifiers of the nodes that caused the RIC to fail.

```
try (Session session = driver.session()) {
  for (int id : listIdEntity) {
    String query = "MATCH␣(node:" + firstEntity +
    ")-[rel:" + tag + "]->(:" + secondEntity +
    ")␣WHERE␣node.id=␣" + id + "␣RETURN␣rel";
    StatementResult result = session.run(query);
    if (!result.hasNext()) {
      if (!idNodeFail.contains(id)) {
        idNodeFail.add(id);
      }
    }
  }
}
```

The continuation of this method is almost identical to the one for the classical relationship.
If the RIC is valid, we will check the condition and obtain the final result of the RIC. After that, we will check the actions with the *"actionTag"* method and finally return the *"listResultRIC"* list containing the global information on the RIC.

Bi-directionality :

The principle of bidirectional relationships is really simple. We are just going to call the unidirectional method twice, from left to right then from right to left. We will therefore have to recuperate the result of the two relationships and compare them to obtain the final result.

For example :

```
Entity1.Attribute1 <-> Entity2.Attribute2
```

This code will be equivalent to :

```
Entity1.Attribute1 -> Entity2.Attribute2
AND
Entity2.Attribute2 -> Entity1.Attribute1
```

Cardinality :

The methods for cardinalities work like the classical methods, except that we will also test the number of elements.
In the case of Classic relationships, we will check that the number of elements in the specified attribute of the node is greater than the minimum cardinality and smaller than the maximum cardinality.
In the other case, for Tag relationships, we will check that the number of relationships, of the specified type with the nodes of the second entity, is greater than the minimum cardinality and smaller than the maximum cardinality.
In the event that the cardinalities are not respected, the RIC will be considered invalid.

Action

Now we must add the concept of action. Actions will have different behaviours depending on whether you are in a Classic relationship or a Tag relationship, therefore they require two distinct methods.

We will explain how the *"Add Info"* and *"Delete"* actions work in both cases and we will illustrate this with a piece of the corresponding code.

Classic relationship :

The *"Add Info"* action for Classic relationships corresponds to the code below. As previously mentioned in the section 4.3.1, the goal of this action will be to add an arc when required to create a relationship that will improve the consistency of the dataset.
The following code will check that each value of a node's attribute (*"valueInt"*) corresponds to an identifier of a node of the second entity (*"listIdSecondEntity*). If there is a correspondence with an identifier, we will check if there is already a relationship between the two nodes. Otherwise, we will add this relationship using the *"createRelationship"* method.

```
for(Object obj : listValue) {
  Value value = (Value) obj;
  int valueInt = value.asInt();
  if (listIdSecondEntity.contains(valueInt)) {
    boolean exist = existRelationship(firstEntity, id,
    secondEntity, valueInt, attribute);
    if (!exist) {
      createRelationship(firstEntity, id, secondEntity,
      valueInt, attribute);
      nodeAdd.add(valueInt);
    }
  }
}
```

After that, we have the *"Delete"* action that corresponds to the code below.
In accordance with section 4.3.2, this action will aim to delete the values of the attribute that do not refer to an existing node.
We will take the values of the first node and check, one by one, that it corresponds to an existing node of the second entity. If this is the case, it is added to the *"updateNodeValue"* list. Once all values have been verified, we will update the first node by replacing its current values with the values from the *"updateNodeValue"* list.

```
for(Object obj : listValue) {
  Value value = (Value) obj;
  int valueInt = value.asInt();
  if (listIdSecondEntity.contains(valueInt)) {
    updateNodeValue.add(valueInt);
  } else {
    nodeDelete.add(valueInt);
  }
}
updateNodeProperty(firstEntity, idNodeFail,
updateNodeValue, attribute);
```

Tag relationship :

For the Tag relationship, we will do the opposite of the classical relationships. Starting from a specific node of the first entity. We will check, for each node of the second entity one by one, if there is a relationship with the node of the first entity. If one exists, the value of the second entity is added to the *"nodeValue"* list.
After testing each node of the second entity, we will update the first node by replacing its current values with the values from the *"nodeValue"* list.
The *"nodeAdd"* list is used to know precisely which values have been added.

```
for (int idSecond : listIdSecondEntity) {
  boolean relationExists = existRelationship(firstEntity ,
  idFirst , secondEntity , idSecond , relationTag );
  if (relationExists) {
    ArrayList<Object> resultValue = getValue(idFirst ,
    firstEntity , relationTag );
    ArrayList<Integer> listValue = new ArrayList<>();
    for (Object obj : resultValue) {
      Value value = (Value) obj;
      listValue.add(value.asInt());
    }
    if (!listValue.contains(idSecond)) {
      nodeAdd.add(idSecond);
    }
    nodeValue.add(idSecond);
  }
}
updateNodeProperty(firstEntity , idFirst , nodeValue ,
relationTag );
```

Finally, we will finish this part with the “Delete” action for Tag relationships. The code below is quite simple : we will have the *“nodeFail”* list which contains the identifiers of nodes that have not respected the RIC and which we will have to check again.

We want to know if the RIC failed because the node in question has no relationship with nodes of the second entity, or did it fail because of cardinalities. To do this, we will check again that it has at least one relationship with a node of the second entity. Otherwise, we will be able to delete this node from the dataset.

```
for (int idNodeFail : nodeFail) {
  boolean exist = false;
  for (int secondId : listIdSecondEntity) {
    exist = existRelationship(firstEntity , idNodeFail ,
    secondEntity , secondId , relationTag );
    if (exist) {
      break;
    }
  }
  if (!exist) {
    deleteNode(firstEntity , idNodeFail);
  }
}
```

Condition

Finally, it was time to add the conditions. Only one version is sufficient for all relationships as the conditions are not dependent on the type of relationship of a RIC.

A hypothesis made on the conditions is that the values tested in the condition are unique values, that means no arrays or lists, and their type must be either String, Integer or boolean. In addition, when the attribute of a condition refers to an attribute in the dataset that has a list or array value, each element of that list or array must comply with the condition to be valid.

The method to check the conditions is quite simple but very long because it requires a lot of verification. That is why we will not show the code of this method in the report.
This method will receive a list including the conditions. For each of them, it will search the value of the attribute in the database. The next step will be to determine which is the relational operator so that the value of the condition can be compared with the value of the database. In the end, the result of each condition will be stored in a list that we will identify by its position in it.

The last step will be to obtain the result of the global condition on the basis of the results calculated for each of the subconditions and their logical operator. Therefore we have developed a recursive method to calculate the final condition :

```
private static boolean recursiveCheck (ArrayList<Boolean>
valueCondition , ArrayList<String> operatorCondition ,
int size) {

  boolean result = false;
  boolean valueRecursive = false;
  size = size - 1;

  if (size == 0) {
    result = valueCondition.get(size);
  } else {
    String operator = operatorCondition.get(size -1);
    if (operator.equals(”AND”)) {
      valueRecursive = recursiveCheck(
      valueCondition, operatorCondition, size);
      result = (valueCondition.get(size) && valueRecursive);
    } else if (operator.equals(”␣OR”)) {
      valueRecursive = recursiveCheck(
      valueCondition, operatorCondition, size);
      result = (valueCondition.get(size) || valueRecursive);
    }
  }
  return result;
}
```

The purpose of this is to take the result of the last subconditions in the list, to take the logical operator of the preceding condition and to call the recursive method with the remaining subconditions of the list.
This will allow us to calculate the subconditions from left to right and return the result of the global condition.

5.2.4 Model-To-Text transformation with Acceleo

This step aims to build the Acceleo code containing both the Java methods developed previously and the Model-to-Text transformation of RICs. To do this, we will need to access the RIC model containing the RICs that have been written by the user.
An example of an XMI file can be found in the Appendix (Appendix B.1 on page 103), so we can see its structure. We have the datasource that corresponds to the connection information, as well as the RICs developed with their properties.

Template

Now we can start writing our code for Acceleo. The first thing to do is to reference the module of our file, that means the URI of the model linked to the XMI file on which we will base to generate the objects of this XMI file.
In our case, it will be : *"http://www.xtext.org/stage/ricdsl/RicDsl"*.

Then, we can design our template with which we will define the creation of our "TransformationJavaApi.java" file. It is also necessary to indicate the package in which the file will be generated, to avoid any compilation error when running the generated file with Java.

```
[module neo4JApiJava('http://www.xtext.org/stage/ricdsl/
RicDsl')/]

[template public neo4JApiJava(aRoot : RicDSL)]
[comment @main /]

[file ('TransformationJavaApi.java', false, 'UTF-8')]
package ['org.eclipse.acceleo.dsl.uml.gen.java.files;'/]


// Java Code //


[/file]
[/template]
```

"aRoot" is the main element of our XMI file that will contain all RICs. We will therefore make a *For* loop on all these elements to process the RICs of the file one by one.

```
[for (ric:RIC | aRoot.rics) before('\n') separator('\n')]

// RIC processing

[/for]
```

Now that we have our RICs one by one, we will be able to obtain the information we are interested in.
Thanks to the code below, we will now determine what type of relationship it is through conditional clauses. In this code, we will go through the RIC : *ric.relationship1.relationship.**TYPE*** . At this point in the XMI file, the RIC will have four variables corresponding to the four types of relationships :

1. ONE_WAY
2. TWO_WAYS
3. TAG_ONE_WAY
4. TAG_BI

Three of them will have a null value while the last will have a not null value. The one with the not null value will correspond to the type of RIC relationship.

```
[if (not ric.relationship1.relationship.ONE_WAY->iterate
(ch; acc:String=''|acc+ch).matches(''))]
listResultRIC = queryOneWay(”[ric.ricId/]”,
”[ric.relationship1.source.entityName/]”,
”[ric.relationship1.target.entityName/]”,
”[ric.relationship1.source.attribute.attributeName/]”,
”[ric.relationship1.target.attribute.attributeName/]”,
 action,
 condition);
[/if]
```

The conditional clause allows us to know which variable is not null thanks to *“not”* and *“.matches(' ')”*. The element “$- > iterate(ch; acc : String ='' |acc + ch)$” will allow us to translate the value of the XMI file into a String that we can then compare to “.matches(' ')” to know if the value is null or not.

In this example, we only test for the *“ONE_WAY”* relationship but we repeat this in the code for the other three types of relationships. We will also test the cardinalities for each one to get our 8 types of relationships.

After determining the type of RIC relationship, the corresponding Java method can be called. In this case, we will call the method “queryOneWay” which will need the RIC information as arguments. We will have the RIC identifier, the entity of the first node, the entity of the second node, the attribute of the first node, the attribute of the second node, the action and the condition. These last two will be explained later.

The principle of actions is the same as for relationships. We have to find out which of the variables, *"ADD_INFO / DELETE / INFO"*, is not null.
The actions in the XMI file are located : *ric.action.**TYPE*** .

```
[if (not ric.action.ADD_INFO->iterate(ch;acc:String
=''|acc+ch).matches(''))]ADD_INFO[/if]
[if (not ric.action.DELETE->iterate(ch;acc:String
=''|acc+ch).matches(''))]DELETE[/if]
[if (not ric.action.INFO->iterate(ch;acc:String
=''|acc+ch).matches(''))]INFO[/if]
```

The conditions contain several elements, so we have chosen to keep them in a list. In addition, there may be several subconditions, so we must make a "For" loop to get them.
The "buildCondition" method will therefore create the list containing all the information about the conditions : the entity, the attribute, the relational operator, the value and the logical operator.

```
[for (cond:Condition | ric.ricCondition) separator('\n')
after('\n')]
condition = buildCondition( "[cond.ent.entityName/]",
"[cond.ent.attribute.attributeName/]", [cond.relationOp],
[cond.value], [cond.logicalOp]);
[/for]
```

Configuration

The entire code presented in the previous section corresponds to what will be in the Acceleo file. We now need to run it to produce the transformation that provides the final Java code.
To do this we must launch the file with Acceleo and make some settings, as shown by Appendix B.2 on page 104. We will have to configure the location of the XMI file, containing the RICs and the desired location of the generated file.

5.2.5 Generate results

This last section of the Implementation part will be devoted to the generation of a file containing the results of each RIC executed.

JSON

JSON technology, also known as *JavaScript Object Notation*, is a data exchange format which is suitable for both humans and machines. Its purpose will be to store information in text form.
It keeps the data in the form of a pairs, represented by a name and a value. Each element is called “Object” and corresponds to an unordered set of pairs, starting with a left brace '{' and finish by a right brace '}'.
The stored values can be of different types such as String, number, boolean, array or again Object.
The API chosen for our project is *JSON-simple* which is quite easy to handle.

Application

The creation of this result file is a simple method that will take as a parameter the RIC with its result to be able to write it into the file.
First, we will check if the file *“RICResult.json”* already exists. If not, we will create it. After that, we will take each RIC and process them one by one.

The important information that will be found in the result file is :

- The validity of the RIC.
- Information on nodes that do not comply with the RIC.
- The action, if there is one.
- Changes produced by the action on the dataset.
- The global condition, if there is one.
- The validity of each of the conditions.

Finally, it is possible to create a JSON object for each of them with the following code :

```
JSONObject node = new JSONObject();
node.put(key, value);
```

Once the final result of a RIC is built, it is written to the file.

File representation

An example of a generated result can be found in the Appendix (Appendix C.1 on page 105). The structure of the JSON file corresponds well to the information presented in the previous section. As the elements of a JSON file have no order, we have redesigned them in the Appendix to have something readable and clear to present.

Chapter 6

Taxonomy of Queries

This chapter aims to explain how the *JCypher API* and *Java Driver API 1.7* queries looks like for each kind of component described in the Chapter 4.
That chapter **Design** explains a theoretical view of a RIC, while this one will provide a more practical view about how a RIC or a set of RICs will be verified in a *Neo4j* database. This will also be helpful for the researchers who will be working on this subject, in particular for the **Future works** (Chapter 9).

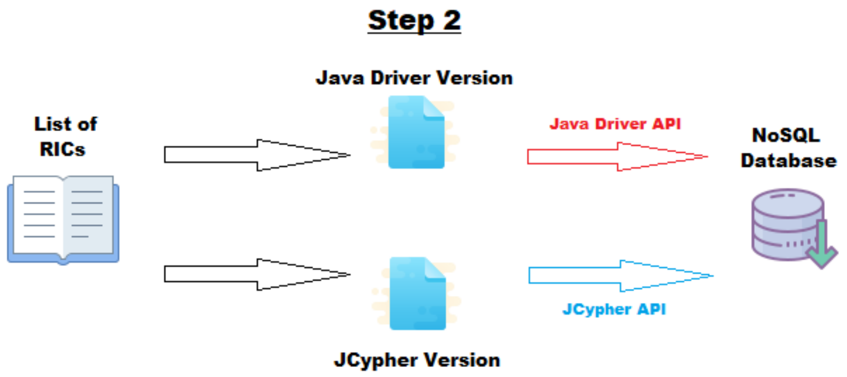


Figure 6.1: Representation of the two APIs in our solution

In Figure 6.1, we can see how our second step of the methodology works schematically. We have as input the list of RICs from the first step and then we have two identical versions of the code that will handle these constraints.
Depending on the version, we will use the corresponding API to perform database queries.
The purpose of this chapter will be to analyze the functions that query the database and to see how they can be built in each of the two versions.
In the future, this could also allow the user to choose which version he wants to use.

6.1 Getting the value of an attribute for all nodes of a specific entity

This query is used for *Classic relationships* whose purpose is to obtain the values of a given attribute for all nodes of a specific entity.

For example :

```
Entity1.Attribute1 -> Entity2.Attribute2
```

In this case, the objective is to verify that all values of the *"Attribute1"* property of an *"Entity1"* node correspond to a value present in a *"Attribute2"* attribute of one of the *"Entity2"* nodes. This query will retrieve the values present in the *"Attribute2"* attribute of all *"Entity2"* nodes.

JCypher API query

```
MATCH.node(node).label(entity),
RETURN.value(node)
```

This query will return all nodes corresponding to the desired entity. We can then, in the Java code, use a *"getProperty"* method specific to this API, to obtain the value of the attribute that interests us for each of the nodes.

Java Driver API query

```
"MATCH␣(node:" + entity + ")" +
"RETURN␣node." + attribute + "␣AS␣Attribute"
```

With this API, we can directly return the value of the attribute for each node.

6.2 Getting the value of an attribute for a specific node

This query will need three parameters : the entity and the identifier of the specific node as well as the attribute whose value we want to return.

JCypher API query

```
MATCH.node(node).label(entity),
WHERE.valueOf(node.property("id")).EQUALS(id),
RETURN.value(node)
```

As the previous one, this query returns the entire node and we must get the value of the attribute in the Java code.

Java Driver API query

```
"MATCH␣(node:" + entity + ")" +
"WHERE␣node.id=␣" + id +
"RETURN␣node." + attribute + "␣AS␣Attribute"
```

6.3 Getting all nodes from a specific entity

The purpose of this query is to return all nodes of a specific entity and then obtain their identifier and store it in the Java code. As a result, we will have a list with all the node identifiers for the desired entity.

JCypher API query

```
MATCH.node(node).label(entity),
RETURN.value(node)
```

As the previous one, this query returns the entire node and we must get the value of the identifier in the Java code.

Java Driver API query

```
"MATCH␣(node:" + entity + ")" +
"RETURN␣node.id␣AS␣ID"
```

6.4 Checking a one-way Tag relationship without cardinality

The purpose of this query will be to verify that there is at least one “Tag” relationship for each node of the first entity with a node of the second entity, which will validate the RIC.
To do this, we will return the existing *“Tag”* relationships between the two entities and check, in the Java code, that the result of this query is not “null”.

JCypher API query

```
MATCH.node(nodeFirst).label(firstEntity).relation(rel)
.out().type(tag).node(nodeSecond).label(secondEntity),
WHERE.valueOf(nodeFirst.property("id")).EQUALS(id),
RETURN.value(rel)
```

Java Driver API query

```
"MATCH␣(node:" + firstEntity + ")-[rel:" + tag + "]->(:" +
secondEntity + ")" +
"WHERE␣node.id=␣" + id +
"RETURN␣rel"
```

6.5 Checking a one-way Tag relationship with cardinality

This query will have the same purpose as the previous one, except that this time we must take into account the cardinalities.
That is why, in this case, the query will return the exact number of “Tag” relationships. We can then check in the Java code that this number is within the cardinality interval to validate the RIC.

JCypher API query

```
MATCH.node(nodeFirst).label(firstEntity).relation(rel)
.out().type(tag).node(nodeSecond).label(secondEntity),
WHERE.valueOf(nodeFirst.property("id")).EQUALS(id),
RETURN.count().value(rel).AS(nCount)
```

Java Driver API query

```
”MATCH␣(node:” + firstEntity + ”)−[rel:” + tag + ”]−>(:” +
secondEntity + ”)” +
”WHERE␣node.id=␣” + id +
”RETURN␣COUNT(rel)␣AS␣Number”
```

6.6 Existing relationship

This query will check if there is a relationship of a specific type between two nodes.
This query will need the entity and identifier of each node as well as the type of relationship.
If the result of the query is not null, we can say that there is a relationship between the two.

JCypher API query

```
MATCH.node(nodeFirst).label(firstEntity).relation(rel)
 .out().type(tag).node(nodeSecond).label(secondEntity),
WHERE.valueOf(nodeFirst.property(”id”)).EQUALS(firstId)
 .AND().valueOf(nodeSecond.property(”id”)).EQUALS(secondId),
RETURN.value(rel)
```

Java Driver API query

```
”MATCH␣(nodeFirst:” + firstEntity + ”)−[rel:” + tag + ”]−>
(nodeSecond:” + secondEntity + ”)” +
”WHERE␣nodeFirst.id=␣” + firstId + ”␣AND␣nodeSecond.id=␣” +
secondId +
”RETURN␣rel”;
```

6.7 Updating a node

In the case of actions, it may be necessary to add or remove values from an attribute of a node.
This query will update the node with its new value after executing an action.

JCypher API query

```
MATCH.node(node).label(entity),
WHERE.valueOf(node.property(”id”)).EQUALS(id),
DO.SET(node.property(attribute)).to(value)
```

Java Driver API query

```
”MATCH␣(node:” + entity + ”)␣” +
”WHERE␣node.id=␣” + id +
”SET␣node.” + attribute + ”␣=␣” + value +
”RETURN␣node”;
```

6.8 Creating a relationship

This query will create a relationship between two predefined nodes.

JCypher API query

```
MATCH.node(nodeFirst).label(firstEntity),
MATCH.node(nodeSecond).label(secondEntity),
WHERE.valueOf(nodeFirst.property(”id”)).EQUALS(firstId)
 .AND().valueOf(nodeSecond.property(”id”)).EQUALS(secondId),
CREATE.node(nodeFirst).relation().out().type(tag)
 .node(nodeSecond)
```

Java Driver API query

```
”MATCH␣(nodeFirst:” + firstEntity + ”),␣” +
”(nodeSecond:” + secondEntity + ”)␣” +
”WHERE␣nodeFirst.id=␣” + firstId +
”AND␣nodeSecond.id=␣” + secondId +
”CREATE␣(nodeFirst)−[:” + tag + ”]−>(nodeSecond)”;
```

6.9 Deletion of a node

In the case of a “DELETE” action, we may have to delete a node from the dataset. This query will allow to delete a specific node using its entity and identifier.
We will use *“DETACH DELETE”* instead of a simple *“DELETE”* to manage cases where the node still has relationships.

JCypher API query

```
MATCH.node(node).label(entity),
WHERE.valueOf(node.property(”id”)).EQUALS(id),
DO.DETACH_DELETE(node)
```

Java Driver API query

```
”MATCH␣(node:” + entity + ”)” +
”WHERE␣node.id=␣” + id +
”DETACH␣DELETE␣node”;
```

6.10 Retrieve the relationships of a node

This query is intended to know all the relationships, incoming and outgoing, of a specific node.
It will be useful in particular when you want to display the global information of a node.

JCypher API query

This query is the only one we have not found a way to reproduce it with JCypher. The problem being to get all the labels of the relationships of a node.
As it is for informational purposes only and does not directly impact the validity of a RIC, this will not prevent the program from functioning properly.

Java Driver API query

Two queries will be necessary to obtain the type of relationship and the targeted nodes.
The first code example will provide all types of relationships related to a node. The second example of code will allow to know, for each type of relationship, the nodes targeted by this relationship.

```
”MATCH␣(node:” + entity + ”)-[rel]-()” +
”WHERE␣node.id=␣” + id +
”RETURN␣type(rel)␣AS␣type”;
```

```
”MATCH␣(node:” + entity + ”)-[rel:” + relation +
”]-(nodeSecond)” +
”WHERE␣node.id=␣” + id +
”RETURN␣nodeSecond.id␣AS␣ID”;
```

6.11 Information about a node

This query aims to obtain the information of a node, in particular each of its properties with the corresponding value.

JCypher API query

As for the first examples of JCypher queries, we will simply return the complete node and then, in the Java code, fetch the attributes one by one thanks to the JCypher API.

```
MATCH.node(node).label(entity),
WHERE.valueOf(node.property(”id”)).EQUALS(id),
RETURN.value(node)
```

Java Driver API query

We will need two queries with the *Jave Driver API*. One to obtain all the properties of the node and one to obtain, property by property, the value.

```
”MATCH␣(node:” + entity + ”)␣” +
”WHERE␣node.id=␣” + id +
”UNWIND␣keys(node)␣AS␣key␣” +
”RETURN␣collect(distinct␣key)”;
```

```
”MATCH␣(node:” + entity + ”)␣” +
”WHERE␣node.id=␣” + id +
”RETURN␣node.” + key + ”␣AS␣” + key;
```

Chapter 7

Experiments

This chapter will represent different kinds of Referential integrity constraints, vary according their type, the cardinalities of the attribute as well as their possible condition and/or action. We will take a simple dataset to which we will apply a set of RICs in order to correct inconsistency errors in the data.
We will start by presenting the initial status of the datastore. Then, the RICs that will be executed, and finally, the final state of the datastore will be displayed. We will end this chapter by reviewing our solution after evaluating it with these tests.

7.1 Initial Data

We will use a more relevant example to analyse RICs than the previous one concerning elections in jurisdiction, with the figure 7.1.
This dataset is composed of *"Actor"* and *"Movie"* nodes. Only one type of relationship is represented : the *"ACTS_IN"* relationship from an *"Actor"* node to a *"Movie"* node. This relationship can have an attribute if it is known, that is the role the actor plays in this film.

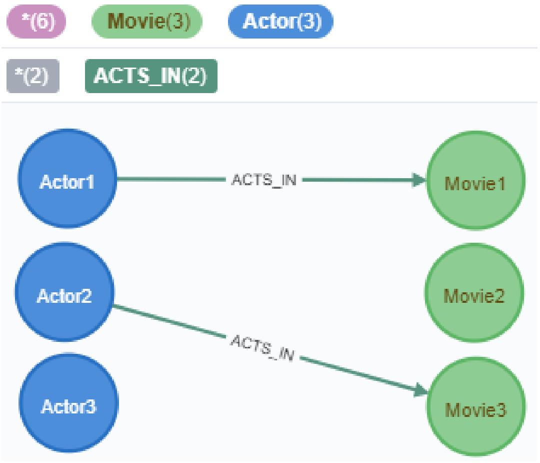


Figure 7.1: Initial dataset

Properties of “Actor” nodes before validation :

Property	Node1	Node2	Node3
name	Actor1	Actor2	Actor3
id	101	102	103
ACTS_IN	[1,2]	[1,19]	
IS_LINKED	true	true	false
numberMovie	2	2	0

Figure 7.2: Experiments : Properties of “Actor” nodes before validation

The *“ACTS_IN”* property contains a list of integers corresponding to the identifier of the movies in which the actor played. *“IS_LINKED”* is a Boolean property indicating that an actor has played in at least one movie. Finally, *“numberMovie”* corresponds to the number of films to which the actor is linked.

Properties of “Movie” nodes before validation :

Property	Node4	Node5	Node6
name	Movie1	Movie2	Movie3
id	1	2	3
ACTS_IN			

Figure 7.3: Experiments : Properties of “Movie” nodes before validation

When we look at these properties, we can see several inconsistencies.
First, the *“Actor1”* node have only one relationship with a node of label *“Movie”* while it has two references in its attribute *“ACTS_IN”*.
Afterwards, the *“Actor2”* node is linked with a node whose identifier is ‘19’ according to its attribute *“ACTS_IN”*. However, no node in the dataset has this one which makes it an incorrect reference. It also has a relationship with the *“Movie3”* node which is not indicated by its property *“ACTS_IN”*.
Lastly, the *“Actor3”* node is not relevant because it has no reference with other nodes.

The program generates the exact result of each RIC in a JSON file. This file, corresponding to the RICs executed in this chapter, has been attached as an Appendix (Appendix C.1 on page 105).

7.2 Example of Referential Integrity Constraints

In this part, we will talk about RICs and their individual impact on the datastores.
Our objective will be to set up a series of RICs to correct the inconsistencies found. Another goal will be to ensure that the RICs also work properly depending on the type of relationship, which is why we will conduct various tests in this section.

Classic relationship without cardinality

```
RIC ric1{
        message: messageRic1
        Actor.ACTS_IN->Movie.id
        in-condition: Actor.IS_LINKED = TRUE
        action: INFO
}
RIC ric2{
        message: messageRic2
        Actor.ACTS_IN->Movie.id
        in-condition:
        action: DELETE
}
RIC ric3{
        message: messageRic3
        Actor.ACTS_IN->Movie.id
        in-condition:
        action: ADD INFO
}
RIC ric4{
        message: messageRic4
        Actor.ACTS_IN->Movie.id
        in-condition: Actor.IS_LINKED = TRUE
        action: INFO
}
RIC ric5{
        message: messageRic5
        Actor.ACTS_IN->Movie.id
        in-condition: Actor.numberMovie < 5
        action:
}
```

The first RIC is a simple Classic relationship where we will check that each value of the *"ACTS_IN"* property of the *"Actor"* nodes corresponds to a value of a *"id"* property of the *"Movie"* nodes. As the *"ACTS_IN"* property of *"Actor2"* has the value '19' but no *"Movie"* node has this value, the RIC will not be validated. In addition, we have a condition that will not be verified since we know that the RIC is not valid. Finally, the *"INFO"* action will have no impact on the dataset.

The second RIC will provide the same result. The difference here is the *"DELETE"* action which will remove the value '19' that is inconsistent since no *"Movie"* node has this identifier.

The third RIC will be validated by the program because no value will be incorrect. However, the *"ADD INFO"* action will have an impact on the dataset because it will create a relationship between *"Actor1"* and *"Movie2"* as the *"Movie2"* identifier corresponds to a value of the *"ACTS_IN"* attribute of *"Actor1"* and the relationship between the two does not yet exist.
Another relationship will be added between *"Actor2"* and *"Movie1"* for the same reason.

The next RIC is the same as the first one but, this time, the result of the relationship will be valid. However, as the relationship is valid, it is now necessary to test the condition and unfortunately, it is not respected by the data so the final result of the RIC will still be invalid. The *"Actor3"* node has the value "False" for its attribute *"IS_LINKED"* which contradicts the condition.

The last RIC of this first part will provide a valid result as the relationship is valid and the condition is satisfied.

Classic relationship with cardinality

```
RIC ric6{
        message: messageRic6
        Actor.ACTS_IN->Movie.id [ZERO..ONE]
        in-condition:
        action:
}
RIC ric7{
        message: messageRic7
        Actor.ACTS_IN->Movie.id [ONE..ONE]
        in-condition:
        action:
}
RIC ric8{
        message: messageRic8
        Actor.ACTS_IN->Movie.id [ZERO..MANY]
        in-condition:
        action:
}
RIC ric9{
        message: messageRic9
        Actor.ACTS_IN->Movie.id [ONE..MANY]
        in-condition:
        action:
}
```

We will now test the four possible combinations of cardinality with the classical relationships.
At the moment in the datastore, *"Actor1"* has two elements in its *"ACTS_IN"* property, *"Actor2"* only one and *"Actor3"* has none.

RIC number 6 will check if each *"Actor"* label node has maximum one element in its *"ACTS_IN"* attribute. This RIC will fail because the attribute *"ACTS_IN"* of *"Actor1"* has two values : [1,2].

The 7th RIC will verify if each *"Actor"* label node has only one element in its *"ACTS_IN"* property.
"Actor1" has more than one element and *"Actor3"* has less so this RIC will fail.

The 8th RIC will be validate because the *[zero..many]* cardinality is always satisfied.

The ninth RIC is the last test of this part and will check, for each *"Actor"* label node, if there is at least one element in its *"ACTS_IN"* property that corresponds to an identifier of a *"Movie"* node. *"Actor3"* has none therefore this RIC will fail.

Tag relationship without cardinality

We are done with Classic relationships, so we will now move on to the Tag relationships with three more RICs in this subsection.

```
RIC ric10{
        message: messageRic10
        Actor-(ACTS_IN)->Movie
        in-condition:
        action: INFO
}
RIC ric11{
        message: messageRic11
        Actor-(ACTS_IN)->Movie
        in-condition:
        action: DELETE
}
RIC ric12{
        message: messageRic12
        Actor-(ACTS_IN)->Movie
        in-condition:
        action: ADD INFO
}
```

The tenth RIC is a simple Tag relationship where we will check that each *"Actor"* node have at least one *"ACTS_IN"* relationship targeting a *"Movie"* node. This relationship is not valid because the *"Actor3"* node has no relationship, therefore the RIC will fail.

A proposal to improve the consistency of this dataset would be to remove the node that has no relationship. That is why we wrote the RIC number 11 which has the *"DELETE"* action and which aims to remove nodes that do not have at least one relationship of type *"ACTS_IN"* with the second entity.
This RIC will not be validated but its action will remove the *"Actor3"* node of the datastore.

The third RIC in this part, *"ric12"*, will be validated because the last two *"Actor"* nodes of the dataset have at least one relationship with a node of the second entity.
The *"Add Info"* action will improve the consistency of the data by adding the identifier of the movie number three in the *"ACTS_IN"* attribute of the *"Actor2"* node because they have an arc (*"ACTS_IN"*) between them.

Tag relationship with cardinality

Now we are going to try the four possible combinations of cardinality with the Tag relationships.
At the moment in the datastore, *"Actor1"* has two *"ACTS_IN"* relationships, *"Actor2"* also has two of them and *"Actor3"* has been removed.

```
RIC ric13{
        message: messageRic13
        Actor-(ACTS_IN)->Movie[ZERO..ONE]
        in-condition:
        action:
}
RIC ric14{
        message: messageRic14
        Actor-(ACTS_IN)->Movie[ONE..ONE]
        in-condition:
        action:
}
RIC ric15{
        message: messageRic15
        Actor-(ACTS_IN)->Movie[ZERO..MANY]
        in-condition:
        action:
}
RIC ric16{
        message: messageRic16
        Actor-(ACTS_IN)->Movie[ONE..MANY]
        in-condition:
        action:
}
```

RICs number 13 and number 14 will each fail because they involve not having more than one *"ACTS_IN"* relationship whereas "Actor1" and "Actor2" have two of them.

The 15th RIC is the same meaning as the eighth, it will be validate because the *[zero..many]* cardinality is always satisfied.

The last RIC of this part, *"ric16"*, is validate because *"Actor1"* and *"Actor2"* have each at least one *"ACTS_IN"* relationship.

Additional tests

In this last part, we will show an example of a multiple condition as well as two bidirectional relationships.

```
RIC ric17{
        message: messageRic17
        Actor.ACTS_IN->Movie.id
        in-condition: (Actor.IS_LINKED = FALSE) AND
        (Actor.numberMovie < 3) OR
        (Actor.numberMovie >= 1)
        action:
}
RIC ric18{
        message: messageRic18
        Actor<-(ACTS_IN)->Movie
        in-condition:
        action:
}
RIC ric19{
        message: messageRic19
        Actor.ACTS_IN<->Movie.ACTS_IN
        in-condition:
        action:
}
```

RIC number 17 represents a multiple condition. The relationship is valid, so we will check the validity of the condition. We divide it into three distinct conditions and obtain respectively as a result :

- Condition 1 : False AND
- Condition 2 : True OR
- Condition 3 : True

By following the order of the conditions, we do : *(Condition 1 AND Condition 2) OR Condition 3.* In other words, we have *(False AND True) OR True*, which results in *True*. The combined result of the relationship and the condition validates the RIC.

The 18th RIC is a Tag relationship but bidirectional. It means that we will execute the RIC from left to right as a Tag relationship, representing the first way, but also from right to left, representing the second way. After that, we will combine the result of the two paths to obtain our final result.
In the first case, from left to right, the result will be positive because each node has at least one relationship with a *"Movie"*. The problem comes from the second part, from right to left, where the three *"Movie"* nodes have no relationship targeting an *"Actor"* node. Therefore, the final result of this RIC will be invalid.

The last RIC of this test is a Classic relationship but bidirectional. The meaning of this is the same as the previous RIC. In this case, the problem comes to the first way, from left to right. As the *"Movie"* nodes have no value in their *"ACTS_IN"* property, we cannot find any correspondence for the values of the *"ACTS_IN"* attribute of the *"Actor"* nodes. That is why this last RIC will fail.

7.3 Data after validation

The Figure 7.4 shows the datastores fixed by RICs. We can see that some relationships have been added to strengthen the consistency of the data. The *"Actor3"* node has finally been removed as it had no reference with other elements of the graph.
As mentioned earlier, the user must be very careful when deleting data from the database.

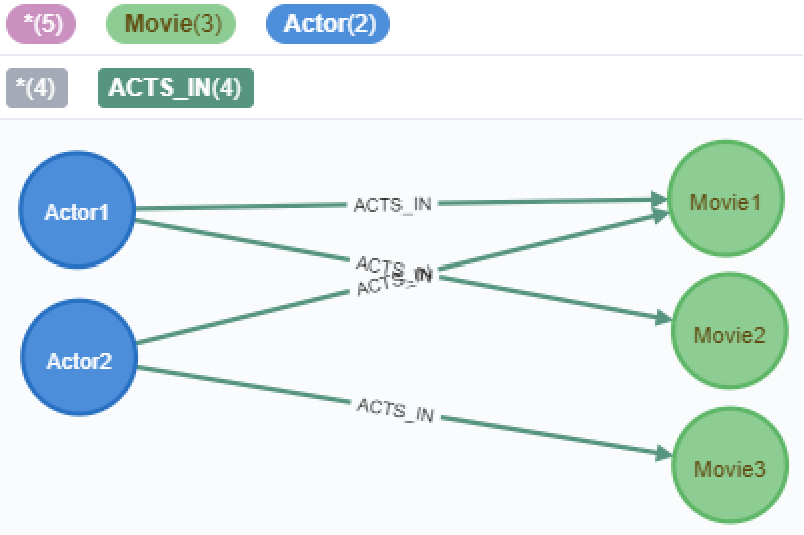


Figure 7.4: Fixed dataset

Properties of “Actor” nodes after validation :

Property	**Node1**	**Node2**
name	Actor1	Actor2
id	101	102
ACTS_IN	[1,2]	[1,2]
IS_LINKED	true	true
numberMovie	2	2

Figure 7.5: Experiments : Properties of “Actor” nodes after validation

Properties of “Movie” nodes after validation :

Property	**Node4**	**Node5**	**Node6**
name	Movie1	Movie2	Movie3
id	1	2	3
ACTS_IN			

Figure 7.6: Experiments : Properties of “Movie” nodes after validation

Finally, we can see that the properties of the *“Actor”* label nodes have also been updated to ensure data consistency.
The *“ACTS_IN”* property has been adapted to match the relationships with the *“Movie”* nodes.
Thanks to the RICs applied to this database, each element linked to another will be linked both by a direct relationship (an edge) and by the value present in its *“ACTS_IN”* property.

7.4 Strengths and weaknesses of our proposal

After these tests, it was time to take a look at our solution.
This section will provide the different strengths and weaknesses of our solution that we were able to observe in this work.

Strengths

- The writing of Referential integrity constraints is quite intuitive thanks to the editor's proposals.
- Code generation is fast and requires only one button press.
- The methodology of our solution is easily understandable as we have seen in these tests.
- The **Project Output** section in the Appendix C.1 allow to the reader to know how the generated results looks like in the *JSON* file and to be able to reuse them later.

Weaknesses

- Not every test possibility have been covered.
- Not all actions could be implemented.
- The editor and the tool to generate the code are not linked, they have not been able to be put together yet. Therefore, it has to be done in two steps, instead of during the same one.

Chapter 8

Conclusion

This chapter will summarise the essential results and our personal opinion about this work.

Summary of the work

The first Chapter **Introduction & Motivation** describes the goal of the work and the frame within the work is carried out.
Information systems become more complex and require more flexibility to address current needs. For that reason, the non-relational databases are nowadays more popular thanks to their heterogeneity and adaptability. Their flexibility is a great help when it comes to processing a very large number of diverse data. The objective of this thesis was to focus on the consistency of the data in the NoSQL datastores, and in particular the Graph-oriented databases. This type of database represents information in the form of nodes and arcs, in particular thanks to the *Neo4J* tool and the *Cypher* language.
The consistency of the data that is not present in NoSQL can be guaranteed thanks to the *Referential Integrity Constraints*, which is a key point in this work. On the basis of all this, we have developed a solution that responds to our needs.

The second Chapter **Background** describes all the needed resources to carry out all the work that had to be done. Starting with an analysis of several related works whose the purpose of which was to learn a little more about what already exists in relation to our subject.
Next, a more conceptual part addressing the subject of *Integrity constraints* and in particular the *Referential Integrity Constraints* by explaining different ways of managing data consistency.
Finally, a technical part dealing with the techniques used like the description of the *Neo4j* technology to manage the Graph-oriented databases as well as its two APIs for Java : *JCypher and Java Driver*. We have also discussed the subject of some *Model Driven Engineering* techniques with some explanations about *Domain Specific Language* and *Model-To-Text Transformation* and their respective tools used in this work. We ended this chapter with an example of how to create a Neo4J database.

After that, comes the Chapter 3 **Methodology** where the two main steps of our solution are described to allow a normal user to understand how to use it. We can draw a parallel between these two steps and the way we developed our solution because they correspond to the concrete syntax definition through *XText* and the semantic part with *Acceleo*.

The following Chapter called **Design** is the fourth and explains the several elements of a *RIC*. We started from a fairly basic representation of a *Referential Integrity Constraint* and explain, step by step, each element we added to it to obtain the final metamodel corresponding to the final representation of a *RIC*. These elements are the kinds of relationship, the cardinalities, the possible actions and the conditions.

The **Implementation** Chapter will aim to developed the code corresponding to the design of *RICs* and their validation. It will again be divided into two parts. First, the definition of the **Ricdsl** language is explained with the advanced features linked to the editor. Then, the implementation around the semantic part is explained with the *Model-To-Text transformation* and the validation.

Afterwards, both APIs used (*Java Driver API 1.7* and *JCypher API*) for *Neo4j* are explained in the Chapter 6 **Taxonomy of queries** with the respective code linked to each feature. Queries built into the code are analysed and explained one by one.
This chapter is useful because it allows us to have an opinion and compare these two APIs. The advantage of *Java Driver API* is that it uses the *Cypher* language directly for its queries. It is therefore intuitive, easier to understand and handle. There is also a lot of documentation and it is easy to find help.
On the other hand, the *JCypher* API is the fluent API for the *Cypher* language which means it is more complex and difficult to handle. Another problem is that the documentation is limited to the author's *GitHub* page and little help can be found on the forums. Nevertheless, its advantages are to propose a prettier, more efficient and shorter code that can be an important choice criterion.
From our perspective, the *Java Driver API* was the most suitable for this short work because it is easily understandable and allows you to quickly get used to using *Neo4J*. However, for use in a larger project, the *JCypher API* is for us the most suitable because it is more efficient and optimised, as long as we allow more time to focus on it.

For a concrete demonstration, a suite of tests is described in the Chapter 7 **Experiments** with a description of each step. We will start with the presentation of a database, then we will write some *RICs* that will aim to solve the consistency problems of this one and we will analyze the final results and impacts on this dataset. The *JSON* output file containing the result of these *RICs* are found in the Appendix C.1. This chapter ends with an analysis of the strengths and weaknesses of our solution.

To end this summary, the **Future works** containing ideas for improvements to our solution are described in the next and last Chapter. The ideas to bring an algorithm for data management, to restructure the validation steps or again to add a new action are discussed there.

Personal opinion

The first part, about the grammar and the Domain Specific Language, was quite interesting because we had to think about how to write the grammar of the **Ricdsl** language. Several meetings with different persons and some talks led to the actual version of this language. Then developing through the *Xtext framework* which was new for us, was quite a good experience too. The final part of this step was to explore some advanced features to customise the editor and to provide a better help for the user was quite fun to do, although the content assistant issue was a bit harder to explore.

For the Model-to-Text transformation part, it was interesting to learn how it worked and to be able to do it for the first time by ourselves. It was a rather vague concept to know how to get a code that met our expectations with just a model.
We choose to use Acceleo for this part because it was a tool used and taught by our receiving institution. Therefore, we were able to benefit from their class slides as well as effective help from our supervisor. Despite this, this tool was a little difficult to learn and required a lot of time to understand. In case of problems, it was not easy to find help on the forums and it was often needed to manage on your own.

The last part, about Graph-oriented datastores and Neo4J, was the most enjoyable part to do. The Graph-oriented version brought a playful side to the NoSQL datastores.
The Neo4J tool is quite simple to use and has clear documentation. In addition, as it is a fairly common tool, there is an easy way to find help on forums. The fact of making two versions of the code with two different APIs allowed to compare these two versions and to make a personal opinion about them.

Chapter 9

Future Works

This chapter will end this thesis by addressing some features that were designed for this work but for which we did not have the time to implement them.

9.1 Integration

The provided editor and the code generation tool developed are not linked together. These are in two different environments. An interesting thing for the future would be to join these two parts following the tutorial *"Creating a UI launcher"* [8] to have a single tool.

9.2 Content assist

As introduced in the Section 5.1.2, several improvements have to be added in the content assistant of the **Ricdsl** language. First of all, in the Section 5.6, the list of all the entities of the database have to be provided by the content assistant to better help the user. Then for the case of the *"complete_Attribute"* method, instead of having a long String with the path of the file like shown in the Figure 5.7, it has to provide the list of the Attributes in the proper Entity. It is planned to explore information from the schema inferred.

9.3 Action improvement

Two functionalities concerning actions had been planned but could not be implemented due to lack of time.

9.3.1 Add cardinality management

First, take into account the cardinalities when developing actions.

In the case of relationships, if the number is not in the interval of cardinalities, the idea would be that *"Add Info"* would add some relationships to reach the minimum cardinality or the *"Delete"* action would delete some to avoid exceeding the maximum cardinality.

The problem was to know which ones to delete or with which nodes to add them. This information is not present in the RIC written by the user.

In the case of Classic relationships, the same idea would be applied to the number of values in the attribute.

9.3.2 Implement the "Delete Cascade" action

Secondly, a *"Delete Cascade"* action has been designed to solve some of the problems of the *"Delete"* action.
When a node is deleted, it can cause a loss of data consistency, in case this node can be part of another RIC. The objective of this action would be to delete, in cascade, the nodes linked to the first one which imply an inconsistency of the data.

This action was not implemented because it was considered quite complex. It is necessary to do many tests to know if the node that will be deleted will impact other RICs. It will be required to determine which other nodes will then be deleted in cascade, and to check for each of them if their deletion will not again impact other RICs.
We are in a situation where a lot of nodes could be deleted from the database, so we have to be careful with this kind of method. Unfortunately, it was not possible to implement this action within the time allowed of our internship.

9.4 Algorithms

The idea of using an algorithm for the Acceleo part was raised several times during the development of the project, instead of processing the data manually like a *"Brute force"*. It would be useful to process the data from the model more efficiently.
First, the *MapReduce* algorithm was mentioned, then we explored *Apache Spark*. Finally, neither was implemented due to lack of time, but it would be interesting to come back to it.

9.4.1 Map Reduce

We started our research on this algorithm by analyzing a paper [9] which discusses about the use of the *MapReduce algorithm* to detect incorrect references in Document-oriented datastores.

The goal of this algorithm is to process the data on large datasets. The main idea is to separate the function **Map** and the function **Reduce**, so to have two different steps :

1. The **map function** iterates on large datasets. It will divide the information into sub-elements and delegate it.
   For each pair (key, value), the map function will give a result list :
   list(keyResult, valueResult).

2. Then the **reduce function** groups and sorts intermediate results. It will forward the result of the intermediate nodes to the parent node : reduce(keyParent, list(intermediateValue)) to calculate the total value.

Let us take a simple example to be complete, based on elections within the jurisdictions, where we have a collection of people that are either *"Male"* or *"Female"* and we want to get the average age of the *"Male"* group and the average age of the *"Female"* group.
The **map function** give each person's age with a (gender:age) couple for each. The **reduce function** will then compute the values average to have a final result of this form : (male, averageMaleAge) and (female, averageFemaleAge) where *"averageMaleAge"* and *"averageFemaleAge"* are respectively the males average age and the females average age.

The goal will be to reduce the number of data that will be compared with those in the database. This algorithm was abandoned because it was not considered relevant and useful enough for this work.

9.4.2 Apache Spark

Therefore we tried to find another solution to optimize the comparison of data with the database and that is why we took an interest in Apache Spark.
Spark [22] is an open-source framework for distributed computing. Its goal will be to process large-scale data by performing complex analyses.
An important advantage [16] compared to other algorithms, and in particular *MapReduce*, is its very high computing speed.
However, the main interest of Spark is its use of Resilient Distributed Datasets (RDD) to apply parallel processing [20] within clusters or computer processors. It uses a cluster manager to coordinate the work. A cluster is a set of computers connected and coordinated with each other to process data and calculate.

This solution for managing the data can be explored in more detail in future work.

9.5 Improve the validation process

Finally, we will finish this chapter by talking about the structure of RIC validation. Currently, we check the validity of an RIC, then the condition and finally the action.

A better structure could be to adapt the validation according to the presence of an action. In case there is no action, the structure would be similar to what we have now : check the RIC and if it is valid, check the condition.
If there is an action, it would be necessary to verify twice the validity of the RIC. Once at the beginning to know the failed nodes and once after the action to have the new result of the RIC after updating the database with the action.

The structure should be : Validation -> Action -> Second validation -> Condition (if there is one and if the RIC is valid).

Bibliography

[1] Michael Blaha. Referential integrity is important for databases, 2005.

[2] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[3] Carlos Javier Fernández Candel. University of murcia, faculty of computer sciences. course : Model-driven software development. practical work : Model-to-text transformations, 2018.

[4] SQL Check Constraint. http://www.sqltutorial.org/sql-check-constraint. Accessed: 2019-05-21.

[5] SQL Constraints. https://www.tutorialspoint.com/sql/sql-constraints.htm. Accessed: 2019-05-21.

[6] Gwendal Daniel, Gerson Sunye, and Jordi Cabot. Umltographdb : Mapping uml to nosql graph databases, 2016.

[7] V. J. Dindoliwala and R. D. Morena. Comparative study of integrity constraints, storage and profile management of relational and non-relational database using mongodb and oracle, 2018.

[8] Eclipse Documentation for Acceleo. https://wiki.eclipse.org/Acceleo/Getting_Started. Accessed: From 02/2019 to 05/2019.

[9] Kalin Georgiev. Referential integrity and dependencies between documents in a document oriented database, 2013.

[10] Carl Henry and Thibaud Staelens. Thesis : Enforcing foreign key constraints in legacy systems, 2018.

[11] Neo4J : Referential Integrity. http://alronz.github.io/Factors-Influencing-NoSQL-Adoption/site/Neo4j/Data\%20Model/Referential\%20Integrity/#neo4j-data-model-referential-integrity. Accessed: 2019-05-23.

[12] JCypher. http://jcypher.iot-solutions.net/. Accessed: 2019-05-10.

[13] Neo4J. https://neo4j.com. Accessed: 2019-05-10.

[14] Peter Neubauer. Graph databases, nosql and neo4j, 2010.

[15] Harsha Raja. Thesis : Referential integrity in cloud nosql databases, 2012.

[16] Philip Rathle. Cypher for apache spark, 2017.

[17] MongoDB. Database References. `https://docs.mongodb.com/manual/reference/database-references/`. Accessed: 2019-05-21.

[18] Fransisco Javier Bermùdez Ruiz. University of murcia, faculty of computer sciences. course : Model-driven software development. chapter : Creating domain-specific languages, 2018.

[19] Fransisco Javier Bermùdez Ruiz. University of murcia, faculty of computer sciences. course : Model-driven software development. chapter : Model-to-text transformations, 2018.

[20] Hari Santanam. How to use spark clusters for parallel processing big data, 2018.

[21] Martina Sestak, Kornelije Rabuzin, and Matija Novak. Integrity constraints in graph databases : Implementation challenges, 2016.

[22] Apache Spark. `https://spark.apache.org`. Accessed: 2019-05-19.

[23] Xtext Eclipse Support. `https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html`. Accessed: From 02/2019 to 05/2019.

[24] How to validate JSON values in NoSQL with check constraint. `https://lefred.be/content/mysql-8-0-16-how-to-validate-json-values-in-nosql-with-check-constraint/`. Accessed: 2019-05-21.

[25] Wolfgang-Schuetzelhofer. Jcypher : Github repository. `https://github.com/Wolfgang-Schuetzelhofer/jcypher/wiki`. Accessed: 2019-05-10.

Appendices

Appendix A

Advanced features code

A.1 Syntax coloring code

Xtend code related to the section 5.1.2, made to have a personal syntax coloring for the *ricdsl* language.

```
package org.xtext.stage.ricdsl.ui.contentassist

import org.eclipse.xtext.ui.editor.syntaxcoloring.
IHighlightingConfiguration
import org.eclipse.xtext.ui.editor.syntaxcoloring.
IHighlightingConfigurationAcceptor
import org.eclipse.xtext.ui.editor.utils.TextStyle
import org.eclipse.swt.graphics.RGB
import org.eclipse.swt.SWT

class RicDslHighlightingConfiguration implements
IHighlightingConfiguration {
  public static final String KEYWORD_ID = ”keyword”;
  public static final String DEFAULT_ID = ”default”;

        TextStyle textStyle;

  override configure(IHighlightingConfigurationAcceptor acceptor) {
    acceptor.acceptDefaultHighlighting(KEYWORD_ID, ”Keyword”,
    keywordTextStyle());
    acceptor.acceptDefaultHighlighting(DEFAULT_ID, ”Default”,
    defaultTextStyle());
  }
  def TextStyle keywordTextStyle() {
    textStyle = new TextStyle();
    textStyle.setColor(new RGB(0, 0, 85));
    textStyle.setStyle(SWT.BOLD);
    return textStyle;
  }

  def TextStyle defaultTextStyle() {
        textStyle = new TextStyle();
    textStyle.setColor(new RGB(127, 0, 0));
    textStyle.setStyle(SWT.ITALIC);
    return textStyle;
  }
}
```

A.2 Content assistant code

Xtend code related to the section 5.1.2 to provide a better content assistant to the user.

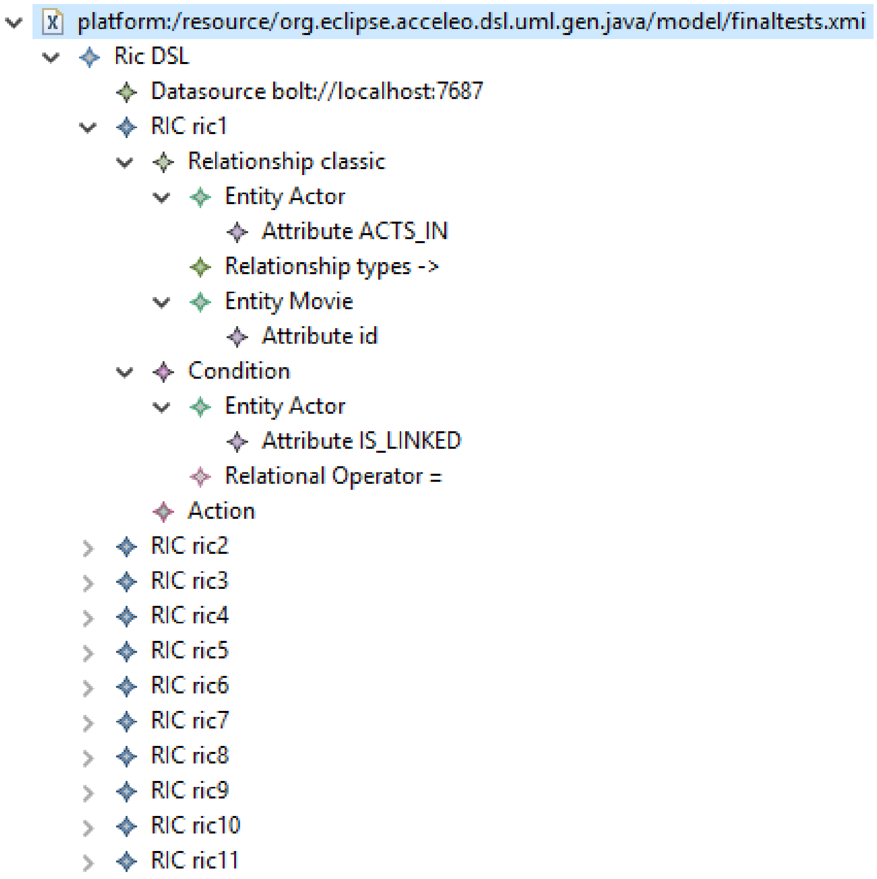
```
/*
 * generated by Xtext 2.16.0
 */
package org.xtext.stage.ricdsl.ui.contentassist

import org.eclipse.xtext.ui.editor.contentassist.ContentAssistContext
import org.eclipse.xtext.ui.editor.contentassist.
ICompletionProposalAcceptor
import org.eclipse.emf.ecore.EObject
import org.eclipse.xtext.RuleCall

/**
 * See https://www.eclipse.org/Xtext/documentation/310_eclipse_support.
 html#content-assist
 * on how to customize the content assistant.
 */

class RicDslProposalProvider extends AbstractRicDslProposalProvider {

  override void complete_Entity(EObject model, RuleCall ruleCall,
  ContentAssistContext context, ICompletionProposalAcceptor acceptor){
  // call implementation of superclass
  super.complete_Entity(model, ruleCall, context, acceptor)
  // compute the plain proposal
  val String proposal = "Here␣should␣be␣provided␣the␣list
␣␣of␣entities␣in␣the␣DataSource"
  // Create and register the completion proposal:
  // The proposal may be null as the createCompletionProposal(..)
  // methods check for valid prefixes and terminal token conflicts.
  // The acceptor handles null-values gracefully.
        acceptor.accept(createCompletionProposal(proposal, context))
  }

  override void complete_Attribute(EObject model, RuleCall ruleCall,
  ContentAssistContext context, ICompletionProposalAcceptor acceptor){
        super.complete_Attribute(model, ruleCall, context, acceptor)
        val String proposal = ”test␣” + context.currentModel
        acceptor.accept(createCompletionProposal(proposal, context))
  }

  //Default information for Neo4j
  override void complete_Datasource(EObject model, RuleCall ruleCall,
  ContentAssistContext context, ICompletionProposalAcceptor acceptor){
        super.complete_Datasource(model, ruleCall, context, acceptor)
        val String proposal = ”DataSource(url=httplocalhost7674,
␣␣␣␣␣␣␣␣usr=neo4j,pwd=password)”
        acceptor.accept(createCompletionProposal(proposal, context))
  }
}
```

Appendix B

Acceleo

B.1 XMI representation

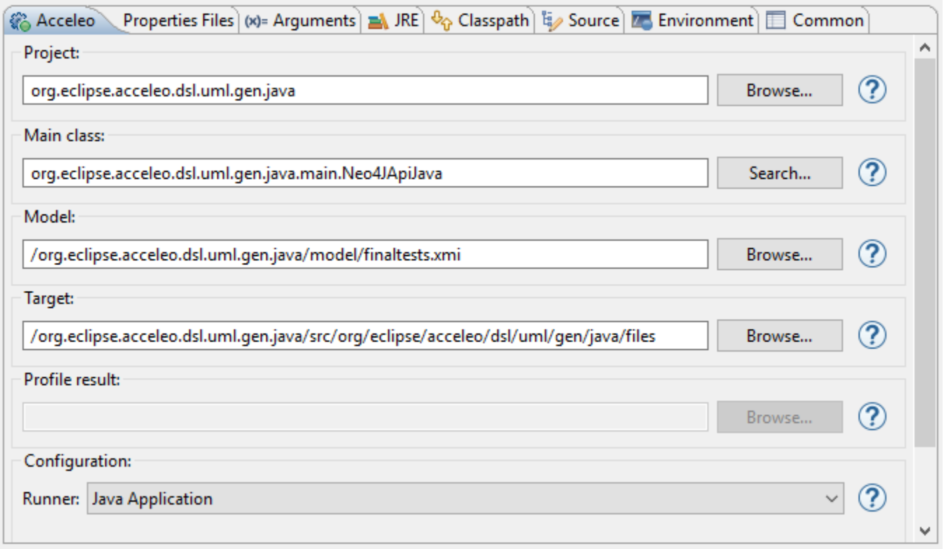
This element represents the structure of the file containing information about RICs.



```
platform:/resource/org.eclipse.acceleo.dsl.uml.gen.java/model/finaltests.xmi
  Ric DSL
    Datasource bolt://localhost:7687
    RIC ric1
      Relationship classic
        Entity Actor
          Attribute ACTS_IN
        Relationship types ->
        Entity Movie
          Attribute id
      Condition
        Entity Actor
          Attribute IS_LINKED
        Relational Operator =
      Action
    RIC ric2
    RIC ric3
    RIC ric4
    RIC ric5
    RIC ric6
    RIC ric7
    RIC ric8
    RIC ric9
    RIC ric10
    RIC ric11
```

B.2 Run configuration

This element represents the configuration required to launch the Model-to-Text transformation in Eclipse.



Project:
org.eclipse.acceleo.dsl.uml.gen.java

Main class:
org.eclipse.acceleo.dsl.uml.gen.java.main.Neo4JApiJava

Model:
/org.eclipse.acceleo.dsl.uml.gen.java/model/finaltests.xmi

Target:
/org.eclipse.acceleo.dsl.uml.gen.java/src/org/eclipse/acceleo/dsl/uml/gen/java/files

Profile result:

Configuration:
Runner: Java Application

Appendix C

Project output

C.1 Generated results in the JSON file

This last element is an example of a file containing the results generated by RICs.

```
[{”ric1”:{
        ”valid”:false,
        ”nodeFail”:{
                ”nodeFail_1”:{
                        ”IS_LINKED”:TRUE,
                        ”ACTS_IN”:[1, 19],
                        ”name”:”Actor2”,
                        ”id”:102,
                        ”numberMovie”:2,
                        ”Relationship : ACTS_IN with ”:[3]
                        }
                },
        ”action”:”INFO”
        }
}]

[{”ric2”:{
        ”valid”:false,
        ”nodeFail”:{
                ”nodeFail_1”:{
                        ”IS_LINKED”:TRUE,
                        ”ACTS_IN”:[1, 19],
                        ”name”:”Actor2”,
                        ”id”:102,
                        ”numberMovie”:2,
                        ”Relationship : ACTS_IN with ”:[3]
                        }
                },
        ”action”:”DELETE”,
        "resultAction":{
                "resultNode_1":{
                        "nodeID":102,
                        "valueDeleted":[19]
                        }
                }
        }
}]

[{"ric3":{
        "valid":true,
        "action":"ADD_INFO",
        "resultAction":{
                "resultNode_1":{
                        "nodeID":102,
                        "relationshipAddWith":[1]
                        },
                "resultNode_2":{
                        "nodeID":101,
                        "relationshipAddWith":[2]
                        }
                }
        }
}]

[{"ric4":{
        "valid":false,
        "action":"INFO",
        "conditionValid":false,
        "resultCondition":{
                "Actor.IS_LINKED = true ":false
                }
        }
}]

[{"ric5":{
        "valid":true,
        "conditionValid":true,
        "resultCondition":{
                "Actor.numberMovie < 5 ":true
                }
        }
}]

[{"ric6":{
        "valid":false,
        "nodeFail":{
                "nodeFail_1":{
                        "IS_LINKED":TRUE,
                        "ACTS_IN":[1, 2],
                        "name":"Actor1",
                        "id":101,
                        "numberMovie":2,
                        "Relationship : ACTS_IN with ":[2,1]
```

```
                              }
                    }
          }
}]

[{"ric7":{
          "valid":false,
          "nodeFail":{
                    "nodeFail_1":{
                              "IS_LINKED":TRUE,
                              "ACTS_IN":[1, 2],
                              "name":"Actor1",
                              "id":101,
                              "numberMovie":2,
                              "Relationship : ACTS_IN with ":[2,1]
                              },
                    "nodeFail_2":{
                              "IS_LINKED":FALSE,
                              "ACTS_IN":[],
                              "name":"Actor3",
                              "id":103,
                              "numberMovie":0
                              }
                    }
          }
}]

[{"ric8":{
          "valid":true
          }
}]

[{"ric9":{
          "valid":false,
          "nodeFail":{
                    "nodeFail_1":{
                              "IS_LINKED":FALSE,
                              "ACTS_IN":[],
                              "name":"Actor3",
                              "id":103,
                              "numberMovie":0
                              }
                    }
          }
}]

[{"ric10":{
          "valid":false,
          "nodeFail":{
                    "nodeFail_1":{
                              "IS_LINKED":FALSE,
                              "ACTS_IN":[],
                              "name":"Actor3",
                              "id":103,
                        "numberMovie":0
                        }
                },
        "action":"INFO"
        }
}]

[{"ric11":{
        "valid":false,
        "nodeFail":{
                "nodeFail_1":{
                        "IS_LINKED":FALSE,
                        "ACTS_IN":[],
                        "name":"Actor3",
                        "id":103,
                        "numberMovie":0
                        }
                },
        "action":"DELETE",
        "resultAction":{
                "resultNode_1":{
                        "idNodeDelete":[103]
                        }
                }
        }
}]

[{"ric12":{
        "valid":true,
        "action":"ADD_INFO",
        "resultAction":{
                "resultNode_1":{
                        "nodeAdd":[3],
                        "nodeID":102
                        }
                }
        }
}]

[{"ric13":{
        "valid":false,
        "nodeFail":{
                "nodeFail_1":{
                        "IS_LINKED":TRUE,
                        "ACTS_IN":[3, 1],
                        "name":"Actor2",
                        "id":102,
                        "numberMovie":2,
                        "Relationship : ACTS_IN with ":[1,3]
                        },
                "nodeFail_2":{
                        "IS_LINKED":TRUE,
                        "ACTS_IN":[2, 1],
                        "name":"Actor1",
                        "id":101,
                        "numberMovie":2,
                        "Relationship : ACTS_IN with ":[2,1]
                        }
                }
        }
}]

[{"ric14":{
        "valid":false,
        "nodeFail":{
                "nodeFail_1":{
                        "IS_LINKED":TRUE,
                        "ACTS_IN":[3, 1],
                        "name":"Actor2",
                        "id":102,
                        "numberMovie":2,
                        "Relationship : ACTS_IN with ":[1,3]
                        },
                "nodeFail_2":{
                        "IS_LINKED":TRUE,
                        "ACTS_IN":[2, 1],
                        "name":"Actor1",
                        "id":101,
                        "numberMovie":2,
                        "Relationship : ACTS_IN with ":[2,1]
                        }
                }
        }
}]

[{"ric15":{
        "valid":true
        }
}]

[{"ric16":{
        "valid":true
        }
}]

[{"ric17":{
        "valid":true,
        "conditionValid":true,
        "resultCondition":{
                "Actor.IS_LINKED = false AND":false,
                "Actor.numberMovie < 3 OR":true,
                "Actor.numberMovie >= 1 ":true
                }
        }
}]

[{"ric18":{
        "valid":false,
        "nodeFail":{
                "nodeFail_1":{
                        "ACTS_IN":[],
                        "name":"Movie3",
                        "id":3,
                        "Relationship : ACTS_IN with ":[102]
                        },
                "nodeFail_2":{
                        "ACTS_IN":[],
                        "name":"Movie2",
                        "id":2,
                        "Relationship : ACTS_IN with ":[101]
                        },
                "nodeFail_3":{
                        "ACTS_IN":[],
                        "name":"Movie1",
                        "id":1,
                        "Relationship : ACTS_IN with ":[102,101]
                        }
                }
        }
}]

[{"ric19":{
        "valid":false,
        "nodeFail":{
                "nodeFail_1":{
                        "IS_LINKED":TRUE,
                        "ACTS_IN":[3,  1],
                        "name":"Actor2",
                        "id":102,
                        "numberMovie":2,
                        "Relationship : ACTS_IN with ":[1,3]
                        },
                "nodeFail_2":{
                        "IS_LINKED":TRUE,
                        "ACTS_IN":[2,  1],
                        "name":"Actor1",
                        "id":101,
                        "numberMovie":2,
                        "Relationship : ACTS_IN with ":[2,1]
                        }
                }
        }
}]
```