Defining Referential Integrity Constraints in Graph-oriented Datastores

Thibaud Masson[1], Romain Ravet[1], Francisco Javier Bermudez Ruiz[2], Souhaila Serbout[2], Diego Sevilla Ruiz[2] and Anthony Cleve[1]

[1]*NADI Research Institute, PReCISE Research Center, University of Namur, Belgium*

[2]*ModelUM Research Group, Department of Computer Science and Systems, University of Murcia, Spain*

Keywords: Referential Integrity, NoSQL, Graph-oriented Datastores, Model-driven Engineering, Domain-specific Languages.

Abstract: Nowadays, the volume of data manipulated by our information systems is growing so rapidly that they cannot be efficiently managed and exploited only by means of standard relational data management systems. Hence the recent emergence of NoSQL datastores as alternative/complementary choices for big data management. While NoSQL datastores are usually designed with high performance and scalability as primary concerns, this often comes at a cost of tolerating (temporary) data inconsistencies. This is the case, in particular, for managing referential integrity in graph-oriented datastores, for which no support currently exists. This paper presents a MDE-based, tool-supported approach to the definition and enforcement of referential integrity constraints (RICs) in graph-oriented NoSQL datastores. This approach relies on a domain-specific language allowing users to specify RICs as well as the way they must be managed. This specification is then exploited to support the automated identification and correction of RICs violations in a graph-oriented datastore. We illustrate the application of our approach, currently implemented for Neo4J, through a small experiment.

1 INTRODUCTION

Since the 1970s, Information Systems (IS) have widely been dependent on relational databases (Wiederhold, 1992). Nowadays, those systems are becoming more and more complex due to the increasingly large amount of data that they need to manage. The enormous growth of data that the world is facing has pushed organisations to migrate to more flexible data storage mechanisms. Therefore, non-relational (NoSQL) datastores became more popular due to their scalability, their flexibility and their ability to easily adapt. NoSQL platforms typically store data without explicit structure, i.e., they are *schema-less*. This ensures a higher level of flexibility during the development process (Leavitt, 2010), and this may also lead to better data access performance (Hendawi et al., 2018). Although in a relational database, a schema clearly describes the characteristics (structures and constraints) of the information stored in the database, it is not the case for a NoSQL datastore (Jatana et al.,)[1].

NoSQL datastores are classified into four distinct paradigmatic families: (1) *key-value*: every data item is stored as a key and a value; (2) *document-oriented*: links every item with a (possibly) complex data structure; (3) *column store*: structures data as columns to optimise queries; (4) *graph-oriented*: uses nodes and relationships to represent data (Nayak et al., 2013). *Graph-oriented datastores* are used to represent references in a simple way, which would be more difficult to model in the relational paradigm. Such a datastore uses nodes, edges and properties to build a graph representing the data. The data items (entities) are represented by nodes, with properties and edges. Node properties correspond to attributes and edges correspond to relationships with other nodes. A graph-oriented datastore can apply CRUD operations (create, recovery, update and delete) on top of a graph data model. This datastore becomes very useful in cases where tree-like or network-like data structures have to be manipulated, e.g., when analyzing social networks. It is no longer necessary to make joins between primary keys, which simplifies the task.

In contrast with relational databases, non-relational datastores do not explicitly handle data integrity constraints, which can prevent users to exploit them. The present work aims to provide a *Model Driven Engineering* (MDE-based) (Brambilla et al., 2012) approach to handle data integrity in NoSQL datastores in general, and in particular, to

[1]Note that we typically use *database* as the term to refer to a relational database, while we use *datastore* to designate a NoSQL database.

manage *Referential Integrity Constraint*s (RICs) in graph-oriented datastores. This approach aims to improve data consistency and therefore data integrity in graph-oriented datastores.

A *Referential Integrity Constraint* is a rule that ensures that the references and relationships between the elements of a datastore are correct. In other words, it prevents users or other applications from impacting negatively the data and the structure within the Database Management System. The aim is to ensure consistency within the database when several data entities are in relationship, like a reference between a `Foreign Key` of a table and a `Primary Key` in another table. In this case, a row in the target table cannot be modified or deleted without impacting on the other table(s) referencing it.

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 to represent artefacts. Moreover, the definition of a Domain-Specific Language (DSL) enables users to work in the proper (domain) abstraction level. Metamodelling and Model Transformations are also MDE techniques for defining and transforming models meaning concepts from different paradigms and different abstraction levels.

In this paper we propose an approach to manage referential integrity constraints in NoSQL, graph-oriented datastores. Most of the time, RICs are used within relational databases but rarely in a NoSQL context. However, using integrity constraints in non-relational datastores allows developers to discover data inconsistencies and to enforce data integrity by fixing those inconsistencies. Therefore, this paper explores this possibility as NoSQL datastores become more and more popular. Our approach makes use of MDE techniques for supporting the definition of RICs and enforcing their validation in a graph-oriented datastore. The main contributions of this work include (1) the definition of a DSL to specify referential integrity constraints (RICs) in a graph-based data model along with an editor for the DSL; and (2) a generative approach that produces a referential integrity management component from the RICs specification. This component automatically identifies RICs violations in the graph-based datastores, and applies data modifications to reestablish consistency if needed, in conformance with the RICS specification.

This paper is organised as follows: Section 2 describes the background needed for a good understanding of our proposal; Section 3 presents our approach, by describing the DSL, its syntax and semantics. A simple proof-of-concept, illustrative experiment of the DSL is presented in Section 4. Finally, we give concluding remarks and anticipate future work in Section 5.

2 BACKGROUND

In this section we will present and describe those concepts required for a better understanding of the solution. Firstly, we will introduce the concept of RICs and some work dealing with its inferring and enforcing. Later, a technological background is given, about graph-oriented datastores and the MDE technologies used in this work.

2.1 Referential Integrity Constraints

The integrity constraints are used to enforce business rules by specifying conditions or relationships among the data. So that any operation that modifies the database must satisfy the corresponding rules without the need to perform any checking within the application. The term integrity with respect to databases includes both database structure integrity and semantic data integrity. Data integrity in the relational context is managed by *primary key*, *foreign key* and *unique* constraints. They corresponds well to the *integrity constraints* for the non-relational domain and, in addition, *check constraints* could also be used to improve data consistency.

In (Blaha, 2019) a *referential integrity* is defined as a protection for the database which makes sure that the references between the data are valid and undamaged. The use of these constraints allows several benefits, such as: improve data quality to preserve references, make the development faster, reduce the bugs amount thanks to better data consistency, enhance the consistency through applications.

Unlike relational databases, where the same structure must be maintained for each set of records (relations or tables), 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.

2.1.1 Inferring Referential Integrity Constraints

In (Meurice et al., 2014), the authors carried out a detailed analysis of the problems posed by the reengineering of a complex information to support RICs, where they found that many of the assumptions generally used in database re-engineering methods do not apply easily. Therewith, they designed a process

and implemented tools for detecting RICs in the context of a real-world problem. Also in (Weber et al., 2014), the authors presented the previous problem in the context of the metaphor of technical debt, used to characterise issues derived from software evolution. The article deals with the problem in the context of databases, and concretely in the technical debt related to database schemas, and the absence of explicit referential constraints (foreign key technical debt).

In (Cleve et al., 2011) the authors use dynamic program analysis as a basis for reverse engineering on relational databases. They show several techniques allowing, in an automate way, to capture the trace of SQL executions and analyse them by applying several heuristics supporting the discovery of referential constraints (implicit foreign keys).

2.1.2 Enforcing Referential Integrity Constraints

In (Georgiev, 2013), the authors address several problems by using foreign keys and some semantic relationships between documents which are lying in the same collection or in different collections. They implement a verification approach which uses the *MapReduce* programming model in document oriented databases.

In (Pokorný and Kovačič, 2017), it is presented the concept of *Integrity constraints* in graph-oriented databases. Various kinds of integrity constraints are developed in the article,including *referential integrity* for checking that only existing entities can be referenced, and each entity should have at least one relationship with an entity of another label to validate a RIC (*foreign key constraint* in relational databases).

2.2 Technological Background

Neo4J [2] is a NoSQL database management system which uses the approach of graph-oriented datastores. It allows data to be represented in a graph as nodes connected by a set of arcs. It presents also *Cypher*, a language that allows to perform queries (on term of nodes and edges) on the datastore and it is composed of a set of clauses, keywords and expressions. Several APIs are provided for executing Cypher. In Java, there are two the main libraries for managing Cypher queries: the Java Driver API for Neo4J and JCypher. The *Java Driver API* enables developers to manage the Neo4J graph database directly from the Java code by writing a Cypher query in a String literal. *JCypher* provides also a fluent Java API that allows to concatenate methods for each clause of a Cypher query.

[2]https://neo4j.com/

MDE techniques have been useful for developing solutions based on code-generation (Kelly and Tolvanen, 2008). Metamodeling and model transformations are the most common techniques, along with the definition of DSLs. Building a DSL involves the definition of its concrete and abstract syntax. Concrete Syntax is normally graphical or textual. *Xtext* is a framework for building DSLs workbenches using a textual syntax. When the grammar is defined, its corresponding *Ecore* metamodel is generated. Also a model serializer to create examples beside a parser and an injector. All these artefacts are provided in the context of an editor. Acceleo [3] is a model-to-text transformation language which is based on the definition of templates that generate the output code by querying the model inputs to be transformed.

3 APPLYING REFERENCTIAL INTEGRITY CONSTRAINTS TO Neo4J

In this section the proposal of the work is described. The grammar of the DSL is defined, detailing the clauses composing the RIC declaration. Later, the metamodel generated by the grammar is depicted. Finally, the semantic of the DSL is presented, showing how the semantic is implemented by using model-to-text transformations and what are the type of Cypher queries generated by the transformation, in order to validate the RICs.

3.1 Proposal

NoSQL datastores lacks of explicitly handle the data integrity. Thus, we propose in this work a solution for managing data integrity in NoSQL graph-oriented datastores by using RICs. The goal of our proposal is therefore to improve the data consistency and integrity.

In order to create the support for defining RICs and applying them on NoSQL datastores, two main artefacts have been defined and implemented. The first artefact is the Domain Specific Language (DSL) for declaring the language intend to define the RICs and the connection to the datastore. The second artefact is a Java code generator to produce the code in order to validate and enforce the declared RICs (providing thus the semantic of the DSL).

Figure 1 depicts the process that developers have to execute using our proposal. In the step (1), developers use the editor provided by the RIC DSL in or-

[3]https://www.eclipse.org/acceleo

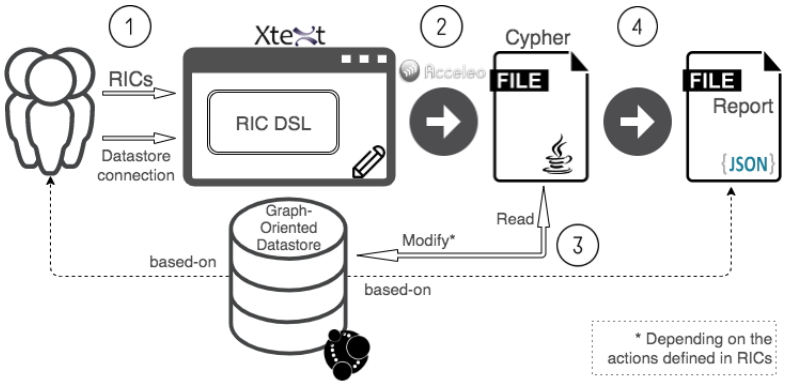

Figure 1: Schema of the proposal.

der to introduce the datastore connection and mainly to define the set of RICs. Developers design the set of RICs based on the graph-oriented datastore in which they want to enforce the constraints. The editor has been made by *XText*. The grammar of the RICs definitions has been designed in order to have an appropriate referring metamodel. In the step (2), the editor is generating automatically the validation code in Java for checking and enforcing (if required) the RICs on the datasource. To do this, it is necessary to build the code in charge of sending the data requests to the graph-oriented datastore. As the datastore that has been selected is *Neo4J*, the queries have been written in the *Cypher* language. The execution of the generated code requires to read the datastore and also modify it, in case that the actions in the RICs are defined for updating the datastore (step 3). In addition, a report file in JSON format is generated for documenting the results of the RICs checking and enforcing on the datastore (step 4).

3.2 Grammar of the DSL

In this section the grammar of the DSL is depicted, starting by defining the main structure and clauses.

```
Datasource(url='connection_url',
           usr='user', pwd='password')
RIC ric_identifier1
    message: info_message
    [ entity1.attributeA -> entity2.attributeB ]
    [ entity1 -(tag)-> entity2 ]
    in-condition: entity1.attributeA = 'value'
    in-fail: [INFO/ADD/DELETE]
RIC ric_identifier2
...
```

The first data that has to be provided is the info about the database connection, in the `Datasource` clause. Then, the set of RICs are listed. Each constraint is defined after the token `RIC`. Firstly, a RIC identifier is declared. Then the message that will be shown in case that it would be required to notify the violation of a constraint in the datasource. Following, the RIC is declared using the appropriate structure of the relationship underlying the constraint: a classical one (`entity1.attributeA -> entity2.attributeB`) that establishes a reference between two attributes of two nodes, or one based on arc tags (`entity1 -(tag)-> entity2`). Then an optional condition could be defined after the token *'in-condition'* for defining an extra-condition to be applied beside the referential constraint. Finally (and also optionally), we could indicate the action to be done in case that a RIC was violated. More details about all this clauses are provided in the following subsections.

3.2.1 Structure of a Relationship

As we described before, we have determined in our language that a RIC can be classified by two main types: *Classic relationship* or *Tag relationship*.

The **Classic relationship** is a RIC based on attributes and is defined with the following syntax: `entity1.attribute1 -> entity2.attribute2`. The goal will be to compare the values of the attribute of a first entity with regards to 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`.

Tag relationship is the reference between two nodes with an arc whose label will be the `Tag`. This type of relationship present the next syntax: `entity1 -(TAG)-> entity2`. Every node belonging to `entity1` should have at least one arc, with the `TAG` name, with an `entity2` element.

Afterwards, we can configure a new type of restriction in the relationship for a RIC. A relationship can be also *bi-directional*, which would mean that the previous definition will be applied for both

ends of the relationship arrow. That is, the relationship will first be analyzed from left to right and then from right to left, as if they were two simple relationships. Thus, a bi-directional relationship will be represented by the next syntax (going from the Classic relationship to the Tag relationship, respectively):
`entity1.attribute1 <-> entity2.attribute2`
`entity1 <-(TAG)-> entity2`

3.2.2 Cardinalities

The cardinality defines the number of occurrences in one entity/attribute which are associated (in a relationship) to the number of occurrences in another. It will be necessary to verify if the value is in the interval specifying the cardinality. In a unidirectional relationship, a cardinality can be added to the destination side of the arrow (for example after `entity2`). While in bi-directional relationships, cardinalities can be added to both sides of the relationship at the same time. It is not mandatory to specify cardinalities but it can be interesting for consistency reasons. An example of their syntax could be `entity1.attribute1 -> entity2.attribute2[MIN..MAX]`

3.2.3 Actions

Actions are defined in order to improve the consistency of the datasource by enforcing the RICs. The actions are checking the validity of a RIC, and they will be performed when a constraint is not accomplished, allowing the user to adapt the content of the database to the constraints defined. 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 also it is the only way to modify the content of the datastore.

This first action is named *Add Info* and it will add some information in the dataset. Its behaviour depends on the relationship type. In the case of a *Classic relationship*, an edge is created when it does not already exist but the constraint is declared. Being a datastore with a classical RIC: `entityRed.attributeRed -> entityBlue.attributeBlue`. With the action *Add Info*, the proposal will add an edge between *nodeA* (of entityRed) and *nodeC* (of entityBlue) only if 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. For the case of *Tag relationship* (`entityRed -(TAG)-> entityBlue`) a checking is made to know if there are relationships, defined by edges, between *nodeRed* nodes and *nodeBlue* nodes. If there is any relationship, the identifier value (in `attributeBlue`) of the *nodeBlue* node is added to the `attributeRed` of the *nodeRed* node.

Summarising, the goal of this action is to keep consistency between nodes and attributes referenced by classical and tag relationships, by means of adding to the nodes the required edges or attribute values, in order to reflect the relationship defined in the RIC.

The purpose of the **Delete** action will be to delete an information or a node violating a RIC. Cautions are needed when this type of actions is used, as it is always risky to delete data from the database. In the case of *Classic relationship* as defined before, each value of the *AttributeRed* are taken. We will check if there is a *nodeBlue* that have this value in its *AttributeBlue*. If no match exists, the value of the *AttributeRed* is removed. For *Tag relationships* as defined before, 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 would be deleted. Another kind of deletion of information has been found out: *Deleting information in cascade*. Even though in the case of *classical relationship* it will not change anything, it will require a special attention 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 its relative nodes in cascade.

The last action is **Info**. The idea here is to show information about a violated RIC. It will return all the information about the node and entity, and its direct relationships with other entities.

3.2.4 Condition

The concept of condition here is to act as an extra-validation for the RIC by checking if the attribute value in the dataset accomplishes with the written condition to validate the RIC. Therefore, not only the classical or tag relationship must be accomplished for not violating a RIC, but also the condition declared in the corresponding clause. The syntax of the condition is described as following: `entity.attribute {RelationalOperator} value {LogicalOperator}`. Although the expression declared for a condition could be very complex, as a simple proof of concept, this work only compares 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 a logical operator, *AND* and *OR*.

3.3 RIC Metamodel

Figure 2 represents the Referential integrity constraint metamodel for the grammar defined in the RICs DSL, after integrating all the elements discussed in the previous grammar section. All classes correspond to the elements of the grammar section, except the `RicDSL` class which represents the root element of the metamodel.

3.4 Semantic of the DSL

The semantic of the RICs DSL has been provided by means of a model-to-text transformation in charge of generating the validation code in accordance with the set of RICs defined. Therefore, a model-to-text transformation translates the RIC declaration in Cypher queries that will be executed on the specified NoSQL datasource. The execution could or not modify the datasource (depending on the action configured) and a JSON file is reporting the status of the validation of each RIC.

3.4.1 M2T Transformation

The model-to-text transformation is done using *Acceleo*. We have considered a hypothesis based on the next assumption about the structure of the nodes of the Neo4J datastore, as simple proof of concept: *each node in the dataset is identified by a property id that is of type Integer*. This attribute will be unique and will allow us to browse through all the nodes in a simple and efficient way. The previous hypothesis can obviously be modified and adapted later according to future needs. The transformation implemented in Acceleo is generating the Java files that implement, in JCypher and/or Java Driver API for Neo4J, the validating queries. Regarding to the clauses defined in the RIC grammar, for a constraint we can combine until three criteria with two possible values each one: (1) the structure of the relationship defining the RIC (Classical or Tag), (2) the directionality of the relationship (unidirectional or bi-directional); and (3) the use of cardinality (yes or not). Therefore, the model transformation is generating *eight* different methods resulting of combining the previous criteria.

We take as an example, for illustrating our explanations, a *Classic relationship without cardinality* between two attributes of different entities. The method must 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. In the algorithm, for each node of the first entity whose identifiers will be kept in a list, it is checked that these values correspond to a value of a node of the second entity. Nodes that do not meet this condition will have their identifier added to a list of failed nodes (named `idNodeFail`). As shown in the pseudocode below, in case where 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 does not contain any condition, we can return the previous result of the validation of the RIC. In the opposite case, we must have to test the condition(s). If the global condition is valid, the RIC will be validated, otherwise it will fail. In addition, if an action had been provided, the action would be tested.

```
if(!idNodeFail.isEmpty()) {
  validRIC=false;
  for(int id : idNodeFail){
    Map<Str,Obj> attNode = info(firstEnt, id);
    nodeFail.add(attNode);
  }
} else {
  if(!listCondition.isEmpty()){
    conditions = checkCondition(listCondition);
    for (Entry<Str,Bool> entry : conditions) {
        if (entry.getKey().equals("result")) {
            validCondition = entry.getValue();
            validRIC = validCondition;
            break;}}
  } else validRIC=true;
}
```

Finally, for the conditions added to the RICs validation, only one version of code is sufficient for all the possible relationships due to the conditions are not dependent on the type of relationship. In order to implement the condition testing, a hypothesis was made: the values checked in the condition are single values (i.e. no arrays or lists), and moreover their type must be either String, Integer or Boolean.

3.4.2 Generated Results

As result of applying the Cypher Java code for validating the set of RICS, a JSON file is reporting the status of the validation. The important information that will be found in the result file is: (1) the validity of the RIC, (2) information about nodes that do not comply with the RIC, (3) the action, if there is one, (4) changes produced by the action on the dataset, (5) the global condition, if there is one, (6) the validity of each of the conditions. For the sake of space reasons, not figures could be included.

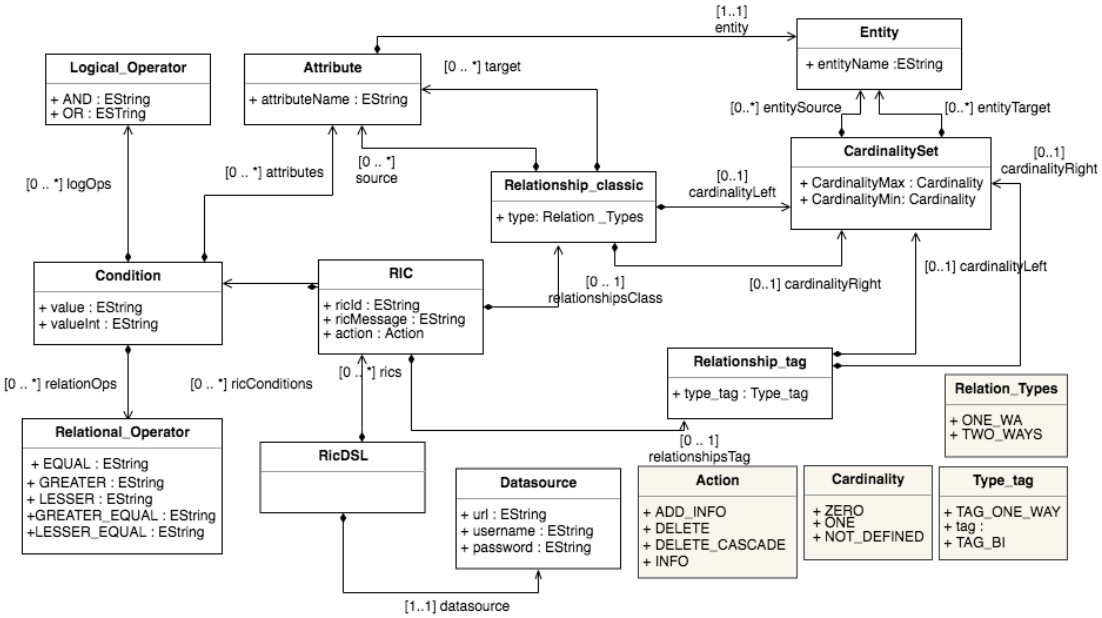

Figure 2: RIC Metamodel generated by XText.

4 EXPERIMENTATION

This section aims to experiment with a simple running example in order to test the different kinds of RICs, vary according their type, the cardinalities of the attribute as well as their possible conditions and actions. The simple dataset used in the experimentation 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. The relationship can have an attribute, that is the role the actor plays in a film (see Figure 3). The properties of nodes before validation are shown in Table 1. The *ACTS_IN* property contains the list of identifiers 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, *numMovie* corresponds to the number of films to which the actor is linked.

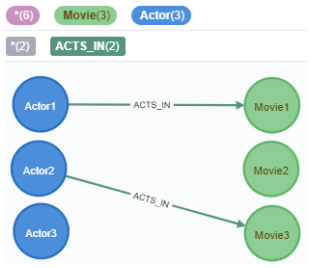

Figure 3: Initial dataset of the experiment.

These properties present several inconsistencies.

Table 1: Properties of nodes before validation.

Property	Node1	Node2	Node3	Node4	Node5	Node6
name	Actor1	Actor2	Actor3	Movie1	Movie2	Movie3
id	101	102	103	1	2	3
ACTS_IN	[1,2]	[1,19]				
is_linked	true	true	false			
numMovie	2	2	0			

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 to a node whose identifier is *'19'* according to its attribute *ACTS_IN*. However, no node in the dataset has this identifier, 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.

We defined a set of RICs in order to validate and correct inconsistencies. Another goal has been to ensure that the RICs also work properly depending on the type of relationship. The test composed of 19 RICs is accessible online [4], and it includes a short description of each RIC.

Figure 4 shows the datastore fixed by the RIC definition. 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 men-

[4] https://cutt.ly/ErqJMtT

tioned earlier, the user must be very careful when deleting data from the database. Finally, we can see in Table 2 that the properties of the *Actor* nodes have also been updated to ensure data consistency. The *ACTS_IN* property has been adapted to match the relationships with the *Movie* nodes. Applying the RICs to this database, each element linked to another will be linked both by a direct relationship (an edge) and also by the value present in its *ACTS_IN* property.

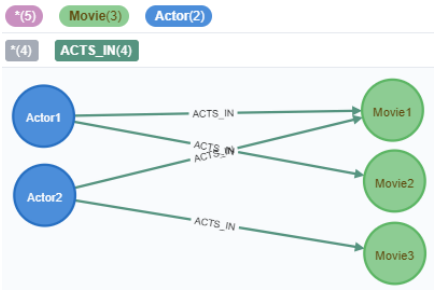

Figure 4: Fixed dataset of the experiment.

Table 2: Properties of nodes after validation.

Property	Node1	Node2	Node4	Node5	Node6
name	Actor1	Actor2	Movie1	Movie2	Movie3
id	101	102	1	2	3
ACTS_IN	[1,2]	[1,2]			
IS_LINKED	true	true			
numMovie	2	2			

5 CONCLUSIONS AND FUTURE WORKS

In this work we have proposed a solution enabling the definition of Referential Integrity Constraints in graph-oriented datastores. For that, a DSL for stating RICs in details has been built. In addition to a code generation for being able to execute the defined RICs in a Neo4J graph-oriented NoSQL datastore. The DSL provided for defining RICs includes several clauses to configure the semantic of the RIC definition: bidirectionality, cardinality, extra conditions and the action to apply in case that a RIC was violated. The implementation has been addressed using MDE techniques, such as the creqtion of a DSL and the implementation of a model-to-text transformation for generating code. The access to the Neo4J datastore relies on the Cypher Java APIs.

In the proposal, we have observed some weakness that have to be improved in future work. First of all, the testing of the tool has to be extended. Not all the possible test cases have been covered. Moreover, the deleting on cascade action has been partially implemented. The editor and the tool to generate the code are not integrated in a unique solution, thus currently it has to be executed separately. Finally, the grammar (concrete and abstract syntax) of the DSL, as well as the semantics must be extended in order to consider other NoSQL datastores (e.g. MongoDB [5]).

ACKNOWLEDGEMENTS

This work was supported in part by the Spanish Ministry of Science, Innovation and Universities, under Grant TIN2017-86853-P.

REFERENCES

Blaha, M. (2019). Referential integrity is important for databases.

Brambilla, M., Cabot, J., and Wimmer, M. (2012). Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182.

Cleve, A., Noughi, N., and Hainaut, J. (2011). Dynamic program analysis for database reverse engineering. In *GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*, pages 297–321.

Georgiev, K. (2013). Referential integrity and dependencies between documents in a document oriented database. *GSTF Journal on Computing (JoC)*, 2:24–28.

Hendawi, A., Gupta, J., Jiayi, L., Teredesai, A., Naveen, R., Mohak, S., and Ali, M. (2018). Distributed nosql data stores: Performance analysis and a case study. pages 1937–1944.

Jatana, N., Puri, S., Ahuja, M., Kathuria, I., and Gosain, D. A survey and comparison of relational and non-relational database.

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley.

Leavitt, N. (2010). Will nosql databases live up to their promise? *Computer*, 43(2):12–14.

Meurice, L. et al. (2014). Establishing referential integrity in legacy information systems - reality bites! In *Proceedings of ICSME 2014*, pages 461–465.

Nayak, A., Poriya, A., and Poojary, D. (2013). Article: Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5(4):16–19.

Pokorný, J. and Kovačič, J. (2017). Integrity constraints in graph databases. *Procedia Computer Science*, 109:975–981.

Weber, J. H. et al. (2014). Managing technical debt in database schemas of critical software. In *2014 6th Int. Workshop on Managing Technical Debt*, pages 43–46.

Wiederhold, G. (1992). Mediators in the architecture of future information systems. *Computer*, 25(3):38–49.

[5] https://www.mongodb.com