Model Driven Reverse Engineering of NoSQL Property Graph Databases

The case of Neo4j

Isabelle Comyn-Wattiau
ESSEC Business School
Cergy, France
wattiau@essec.edu

JackyAkoka
CEDRIC-CNAM & IMT-TEM
Paris, France
jacky.akoka@lecnam.net

***Abstract*— Most NoSQL databases are schemaless. Although they offer some flexibility, they do not have any knowledge of the database schema, losing the benefits provided by these schemas. It is generally accepted that data modelling can have an impact on performance, consistency, usability, and maintainability. We argue that NoSQL databases need data models that ensure the proper storage and the relevant querying of the data. This paper seeks to present and illustrate an MDA-based approach, allowing us to achieve a reverse engineering of NoSQL property graph databases into an Extended Entity-Relationship schema. The approach is applied to the case of Neo4j graph database. We present an illustrative scenario and evaluate the reverse engineering approach.**

Keywords— Reverse engineering, NoSQL graph database, Model-Driven Approach, property graph, Neo4j

I. INTRODUCTION

The volume of data processed by companies has been increasing significantly for the past years. For example, Twitter stores seven terabytes of data every day. Relational database systems have difficulties dealing with these large and unstructured data. In order to handle the problems inherent to massive and unstructured data, companies are increasingly inclined to use NoSQL databases, which become an alternative to traditional SQL database systems. NoSQL databases allow the storage and the processing of huge amounts of unstructured data. They are generally divided into four categories: key-value, document oriented, column-family, and graph databases. In this article, we consider only the graph database category, especially the NoSQL property graph databases.

The design and development of NoSQL databases is mainly empirical. The absence of a schema offers a certain flexibility, but does not allow to benefit from its advantages. In NoSQL databases, the schema is frequently unknown. However it is important to have some information about the structure of the datasets in order to facilitate their processing. The lack of a clear schema design can induce some overhead and complexity, especially for complex queries. Our assumption is that models are needed in order to obtain some level of generality. A logical model, associated with a conceptual model, are relevant, even in the domain of NoSQL databases. In this paper we present an approach based on a model driven reverse engineering of NoSQL property graph databases. Starting from a Neo4j code and using transformation rules, we derive a logical graph model and a conceptual Extended Entity-Relationship (EER) schema.

The rest of the paper is organized as follows. In Section 2 we present a state of the art on modeling NoSQL databases, both forward and reverse engineering. Section 3 describes our model-driven approach (MDA) dedicated to reverse engineering of NoSQL property graph databases. The approach and the transformations rules are illustrated in Section 4. We also present and discuss some evaluation criteria related to the quality of the models obtained. Finally, Section 5 presents some conclusions as well as some perspectives in terms of future research.

II. STATE OF THE ART

Modeling NoSQL databases can be performed using either a forward or a reverse engineering methodology. Modeling NoSQL databases using forward engineering consists in going from a conceptual to a physical data model. Several contributions have been made in modeling document NoSQL databases [1,2]. Some authors propose transformations approach for column store databases [3,4]. As far as key-value databases is concerned, [5] discusses design issues related to key-value data stores. Let us also mention the design process for big data warehouses proposed by [6]. As for graph databases, [7] has presented an UMLtoGraphDB framework. [8] present an approach transforming relational and RDF models to a property graph. Let us mention some contributions on XML database schema generation and extraction [9]. A language for declaring schema evolution operations for extensible record stores and for NoSQL document stores can be found in [10]. Finally, it is worth mentioning contributions related to the transformation of multidimensional data warehouses [11,12].

Reverse engineering is the process of translating the source code of an application into a conceptual model. It allows the elicitation of the missing semantic schemas, recovering the complete specification of a database. Many publications have shown how to generate rich conceptual schemas from instances [13]. But most of them are not related to NoSQL databases [14]. Unlike NoSQL forward engineering, there is little research on NoSQL reverse engineering. [15] present a model-

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

driven reverse engineering approach for inferring the schema of aggregate-oriented NoSQL databases. [16] generate a graph model for mobiles platforms.

To the best of our knowledge, there is no published approach on NoSQL property graph database reverse engineering. This is precisely the purpose of the next section.

III. Our Approach: Model-Driven Reverse Engineering

Reverse engineering of databases allows us to generate logical and conceptual schemas for existing databases. This reverse engineering effort is required to facilitate the understanding, the documentation, and the evolution of databases. Our approach allows us to transform a physical schema described with Neo4j definition language into an EER model, using a graph meta-model as an intermediate step.

To perform the reverse engineering transformations, we adopt a model-driven approach defining different viewpoints of a system, each one represented with specific models (conceptual, logical, and physical). Semi-automatic mappings between these models enable to deploy the system in a specific environment.

A. The reverse engineering approach

The main steps of our reverse approach are: (Step 1) collect all Cypher codes that enabled the Neo4j graph database generation; (Step 2) parse the code to deduce the logical graph using transformation rules; (Step 3) map this logical graph into a conceptual EER schema using transformation rules; (Step 4) complete the resulting schema in particular by adding the cardinalities. Performing this reverse process requires the definition of two meta-models (source and target) presented below. Step 1 is straightforward. The other steps are described below. The underlying principles guiding our mapping process may be summarized as follows: i) elicit the semantics as faithfully as possible; ii) keep track of data instances; iii) factorize properties (attributes and relationships); iv) position attributes and relationships at the relevant level; v) make the process independent of the order of the code's statements.

B. The meta-models

The first meta-model is dedicated to the definition of graph database schemas (Fig. 1). Neo4j supports property graphs. The latter have a set of vertices and a set of edges. Each vertex has a unique identifier, a label, a set of outgoing edges, a set of incoming edges, and a collection of properties. Each edge has a unique identifier, an outgoing tail vertex, an incoming head vertex, a label, and a collection of properties. Reverse engineering implies the abstraction of the graph concepts. We propose to use the labels (node labels as well as edge labels) to structure the graph. Our meta-model extends slightly the one proposed by [7] by adding the set of instances as properties of vertices and edges. Storing the instances along the reverse process enriches our approach by making more inferences possible.

The second meta-model allows us to represent an EER model. The main concepts are: entities, relationships, and attributes. Relationships may be N-ary. EER includes ISA relationships (Fig. 2). In this paper, we only take into account property graphs and not hypergraphs. Thus the relationships resulting from our reverse process are all binary. We assume that relationships are directed since graphs usually contain directed edges. Hence, in the meta-model, relationships between entities are represented by source and target relationships. Our approach generates ISA relationships when nodes contain several labels. Thus, we had to represent these ISA relationships and provide them with id's. To facilitate a roundtrip engineering including both forward and reverse activities, we also store instances at the conceptual level, but only for entities and relationships, and not for attribute values.

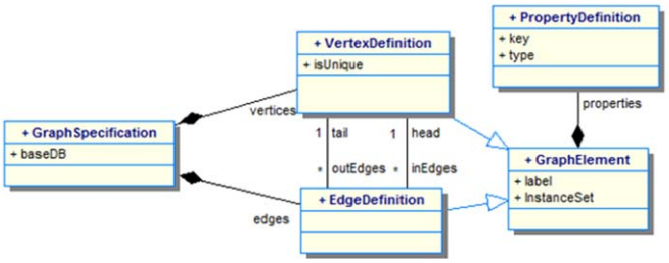

Fig. 1. GraphDB meta-model (adapted from [7])

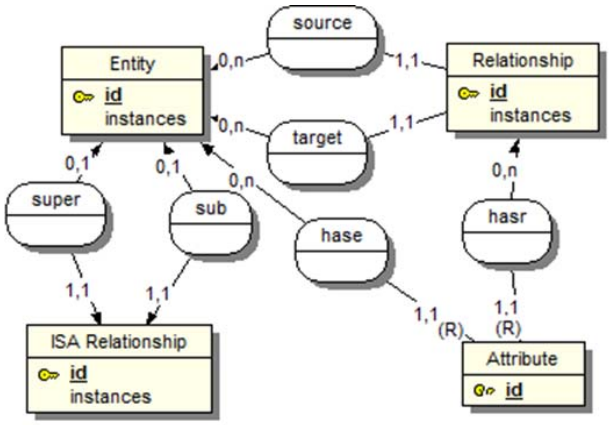

Fig. 2. EER conceptual meta-model

Starting from a Neo4j graph definition set, we first generate an instance of the GraphDB meta-model and then an instance of the EER meta-model. To perform these two steps, we define a set of transformation rules allowing an automatic mapping between the three relevant levels (physical Neo4j database, logical graph database, conceptual EER model). These rules are described in the following sections.

C. From Neo4j database to logical graph database (Step 2)

Let us recall that the creation of a Neo4j graph using Cypher is performed through one or several CREATE statements. As an example, the two following statements perform the insertion of respectively a node and an edge in the graph database: `CREATE(a:b {c:d})` and `CREATE(a-[f]->g.` As an example, `CREATE (shakespeare:Author {firstname: 'William'; lastname:'Shakespeare'})` generates the creation of a node with label `Author`. The statement `CREATE ((shakespeare)- [:BORN_IN]-> stratford)` generates an

edge (label BORN_IN) between nodes shakespeare and stratford. Finally, we want to point out the following syntax: CREATE (billy :User :Author {id:45}). It generates a node billy labelled both Author and User. We will build on such statements to generate ISA links at the conceptual level. Underlying our mapping process is the hypothesis that the labels attached to nodes allow us to group together consistently such instances since they share these labels.

Table 1. Mapping of statement CREATE (a :b {c:d})

Mapping of statement CREATE (a:b {c:d})	b and c exist c belongs to b	b and c exist c does not belong to b	b exists c does not exist	b does not exist c exists	b and c do not exist
Mapping rule number	R11	R12	R13	R14	R15
create v with v.label=b				x	x
add a to v.InstanceSet	x	x	x	x	x
create p with p.key=c			x		x
add p to v.properties		x	x	x	x
propagate a toward generic of v	x	x	x		

We parse these CREATE statements and generate an instance of the GraphDB metamodel. Borrowing from [7], the latter is defined as a tuple GD = (V; E; P), where V is the set of vertex definitions, E the set of edge definitions, and P the set of property definitions that compose the graph. As an example, the statement CREATE(a:b {c:d}) may be mapped using five alternative rules depending on other existing concepts (Table 1). For instance, rule R13 can be expressed as follows:

```
R13: IF      ∃ v ∈ V |v.label=b
     AND     ∀ p ∈ v.properties p.key ≠ c
     AND     ∀ p ∈ P p.key ≠ c
     THEN    let create p in P | p.key=c;
             v.properties=v.properties ∪ {p};
             v.InstanceSet = v.InstanceSet ∪ {a};
             propagateingeneric(a,v)
```

In this case, we suppose that the node v with v.label= b already exists but the property p with key c does not exist. Such property p is thus created and inserted in v.properties. a is added to v.InstanceSet. Moreover, in case where v already contains generic nodes, a is also propagated in all generic nodes of v. CREATE (shakespeare:Author {firstname:'William'}) adds shakespeare to the instances of v, create the property p with p.key=firstname in P, adds p in v.properties and propagates shakespeare in the generic nodes of v if any. Rule R11 is even simpler since it only adds a to v.InstanceSet and propagates a toward generic nodes of v if any.

The rules are based on the statement CREATE(a:b {c1:d1, c2:d2, .., cn:dn}). To ease the understanding of these rules in the paper, we suppose that the physical node a contains only one property value leading to the statement CREATE(a:b {c:d}).

Let's consider another statement frequently found in Cypher code: CREATE (a :b :c {d:e}). The treatment of this kind of statement requires fifteen production rules performing the mapping between the physical and the logical levels (Table 2). The complexity of this case is due to the two labels (b and c) attached to node a. Our abstraction hypothesis is that the label of a node is a set containing all the nodes sharing this label. Thus, a node with two labels materializes the intersection of two sets. In order to reverse this type of node, we have many cases to deal with, depending whether the two labels are new, whether they have already been inserted in intersections with other labels, etc.

As an example, rule R25 tackles the case where the label c is not new but label b is first encountered. Moreover, the property d is also new. R25 can be expressed as follows:

```
R25:   IF      ∃ v2 ∈ V | v2.label=c
       AND     ∀ v1 ∈ V, v1.label ≠ b
       AND     ∀ p ∈ P, p.key ≠ d
       THEN let create v1 in V | v1.label=b;
            let create v3 in V | v3.label=b_c;
            let create v4 in V | v4.label=b&c;
            let create p in P | p.key=d;
          v1.InstanceSet := v1.InstanceSet ∪ {a};
          v2.InstanceSet := v2.InstanceSet ∪ {a};
          v3.InstanceSet := v3.InstanceSet ∪ {a};
          v4.InstanceSet := v4.InstanceSet ∪ {a};
          v4.properties := v4.properties ∪ {p}
```

In this case, we suppose that node v2 labelled c exists already at the logical level but the node v1 labelled b does not exist (and of course the generic and the specific of v1 and v2 do not exist) and property p with p.key=d does not exist. As a consequence, v3 and v4 (corresponding respectively to the generic node and the common subtype of v1 and v2) are created. Moreover, a is added to the set of existing instances of v1, v2, v3, v4 and p is created and added to the properties of v4. Let us note that when a property (labeled d here) appears first as attached to an instance common to two nodes, we attach this property to the common specific node of the two nodes. If, later, another node is inserted sharing the same property but only labeled b for example, the property labeled d will be transferred to the node labeled b at the end of the process (finalization step). As an illustration, let us suppose that the node v1 labeled User already exists but not v2 labeled Author. We suppose also that the property id does not exist at all in the logical graph model. In this case, the statement CREATE (billy :User :Author {id:45}) will fire rule R25 leading to: 1) creating v2, v3, and v4 nodes respectively labeled Author, User_Author (representing the common generic of User and Author), and User&Author (representing the common subtype of User and Author), 2) creates the property p with

`p.key=id` in P, 3) adds `p` in `v4` properties, and 4) adds `billy` in the instance sets of `v1, v2, v3,` and `v4`.

Table 2. Mapping of statement CREATE (a :b :c {d:e})

Mapping of statement (a:b :c {d:e})	b and c exist, b_c and b&c exist				b and c exist, but b_c and b&c don't exist			exists and c does not exi			c exists and b does not exist			b and c do not exist	
	d exists			d does not exist	d exists		d does not exist	d exists		d does not exist	d exists		d does not exist	d exists	d does not exist
	d ∈ b&c	d ∉ b&c: d ∈ b or d ∈ c	d ∉ b&c: d ∉ b and d ∉ c		d ∈ b or d ∈ c	d ∉ b and d ∉ c		d ∈ b	d ∉ b		d ∈ c	d ∉ c			
Mapping rule number	R16	R30	R18	R17	R19	R20	R21	R27	R22	R26	R23	R24	R25	R29	R28
create v3 with v3.label=b_c					X	X	X	X	X	X	X	X	X	X	X
create v4 with v4.label=b&c					X	X	X	X	X	X	X	X	X	X	X
create v1 with v1.label=b											X	X	X	X	X
create v2 with v2.label=c								X	X	X				X	X
create p with p.key=d				X			X			X			X		X
add p to v4 properties			X	X		X	X		X	X		X	X		X
add a to v1,v2,v3 and v4 instances	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Let's consider another statement performing the insertion of an edge in the physical Neo4j database `CREATE ((a)-[:f]->(g))`. This statement may lead to one rule out of two. As an example, rule R32 can be expressed as follows:

```
R32: IF  ∀ e ∈ E | e.label ≠ f
     AND ∃ v1 ∈ V |a ∈ v1.InstanceSet
     AND ∀ v2 ∈ V, (a ∉ v2.Instanceset OR v2 is not
                    a subtype of v1)
     AND ∃ v3 ∈ V |g ∈ v3.InstanceSet
     AND ∀ v4 ∈ V, (g ∉ v4.Instanceset OR v4 is not
                    a subtype of v3)
     THEN let create e in E | e.label=f;
               e.tail:=v1; e.head:=v3;
                e.InstanceSet:= {(a,g)}
```

In this case, we suppose that the edge labeled `f` does not exist yet at the logical level. Then, an edge labeled `f` is created and `(a,g)` is inserted in the instance set of this edge. The second rule R33 is similar to R32 but adds also `e` to E with label `f`.

Table 3. Mapping logical to conceptual

Mapping logical to conceptual	Vertex v	Edge e		
		e.label='ISA'	e.label < > 'ISA'	
			first occurrence of e	additional occurrence of e
Mapping rule number	R51	R52	R53	R54
Entity en with en.id=v.label, properties and instances are transferred	X			
ISA relationship		X		
Relationship r with r.id=e.label			X	
Properties are transferred			X	X
Instances are transferred		X	X	X

For space reasons, we are not able to provide the reader with the whole set of rules. For example, the rules related to case where edge categories also have properties are not described in this paper. This is the same for rules mapping the statements including constraints such as unicity for properties.

D. From logical graph database to EER model (Step 3)

We present below transformation rules allowing us to map a graph description (V, E, P) into an EER model (Table 3). The latter is defined as a tuple (EN, A, R, ISA, SUB, SUPER, SOURCE, TARGET, HASE, HASR) where EN, A, R, ISA denote the set of respectively entities, attributes, relationships, ISA relationships. SUB, SUPER, SOURCE, TARGET, HASE, HASR store the instances of all the so-called relationships of our EER meta-model. SOURCE and TARGET are composed of triples bearing the cardinality of the relationship.

The first rule of the reverse process from logical to conceptual level maps the elements of V into elements of EN. Each vertex belonging to V becomes an entity included in EN. We store also its instances and its attributes. More precisely, it can be expressed as follows:

```
R51: IF ∃ v ∈ V |v.label=b
      THEN let create en in EN |en.id=v.label;
           A:= A ∪ v.properties;
           en.InstanceSet:=v.InstanceSet;
        ∀ a ∈ v.properties, HASE := HASE ∪ {(en,a)}
```

The second rule is related to the reverse engineering of ISA edges. An edge labeled ISA becomes a generalization link between corresponding entities in the EER model (rule R52):

```
R52: IF ∃ e ∈ E | e.label='ISA'
    THEN  let create i in I with i.id=generateid();
        I:=I ∪ {i}; SUPER:=SUPER ∪ {(i,e.head)};
        SUB:=SUB ∪ {(i,e.tail)}
```

As for the other edges, a set of rules allows us to ensure their mappings to the corresponding elements of R. To this end, we need two different rules. The first rule R53 can be expressed as follows:

```
R53: IF  ∃ e ∈ E | e.label ≠ 'ISA'
     AND ∀ r ∈ R, r.id ≠ e.label
     THEN   let create r in R with r.id=e.label;
            r.InstanceSet:=e.InstanceSet;
           SOURCE:= SOURCE ∪ {(r,e.tail,1)};
           TARGET:= TARGET ∪ {(r,e.head,1)};
         ∀ a ∈ e.properties, HASR := HASR ∪ {(r,a)}
```

If it is the first occurrence of an edge from E which is not labeled ISA, it becomes a relationship between corresponding entities. If not, rule R54 presented below applies:

```
R54:   IF ∃ e ∈ E | e.label ≠ 'ISA'
       AND ∃ r ∈ R | r.id = e.label
       AND ∃ en1 ∈ EN | (en1,r) ∈ SOURCE
       AND ∃ en2 ∈ EN | (r,en2) ∈ TARGET
       AND ∃ en3 ∈ EN | en3.label=e.tail
       AND ∃ en4 ∈ EN | en4.label=e.head
 THEN r.InstanceSet:=r.InstanceSet ∪ e.InstanceSet;
       attachsourceofrelationship (en1, en3, en5) ;
       attachtargetofrelationship (en2, en4, en6) ;
         SOURCE:= SOURCE ∪ {(r,en5,'1')};
         TARGET:= TARGET ∪ {(r,en6,'1')} ;
        ∀ a ∈ e.properties, HASR := HASR ∪ {(r,a)}
```

Let `e` be an edge which is not ISA. It represents a link between two categories that we mapped into two entities `en3` and `en4` using rule R51. This link is mapped into a binary relationship `r` between the two entities. If this is not the first occurrence of this link, `r` already exists in the EER model between two entities `en1` and `en2`. We have to check the links between `en1` and `en3` on the one hand and between `en2` and `en4` on the other hand. Four cases may occur: i) `en1` and `en3` are identical, ii) `en1` and `en3` belong to the same branch of an ISA hierarchy, iii) `en1` and `en3` belong to different branches of an ISA hierarchy, iv) `en1` and `en3` are different and are not in the same hierarchy. The same situation arises for `en2` and `en4`. The routine `attachsourceofrelationship` performs the correct attachment of `r` for each of the four cases (the same principle applies to routine `attachtargetofrelationship`). The underlying principles guiding the attachment of `r` are: i) avoiding redundancy and thus representing only one relationship, ii) factorizing as much as possible the common properties, i.e. the relationships, iii) preserving the semantics by attaching `r` at the adequate level of the hierarchy.

E. Finalization step (Step 4)

The main tasks of this step are: i) propagating instances using a bottom-up process through all ISA generalization hierarchies; ii) factorizing properties at the adequate level in the ISA hierarchies; iii) generating the cardinalities. The first two tasks are straightforward. For the last task, the principle is to parse all relationship instances and transform the default '1' cardinality into 'N' as soon as we encounter two or more instances linked to the same instance by the relationship. We present below the rule R71 performing this task for the sources of the relationships. A similar rule is used for the targets.

```
R71:   IF ∃ r ∈ R AND ∃ (i1,i2) ∈ r.InstanceSet
       AND ∃ (i1,i3) ∈ r.InstanceSet
       AND ∃ en ∈ EN | (r,en,'1') ∈ SOURCE
 THEN SOURCE:= SOURCE ∪ {(r,en,'N')} -{(r,en,'1')}
```

Our approach contains many rules for the first step (from physical to logical levels) since we wrote them with the objective of generating the same logical schema whatever the order of the code's statements. Moreover, the condition parts of the rules are mutually exclusive. Therefore, the result does not depend on the firing order of the rules. Regarding the second step (from logical to conceptual levels), for sake of simplicity, in order to avoid the multiplication of similar rules, we first translate the vertices (rule R51) and then the edges (rules R52 to R54). The consequence is that the second step needs to be adapted to the case of continuous flow of data.

IV. ILLUSTRATIVE SCENARIO AND EVALUATION

Starting from the following Cypher code (Fig. 3), we generate the logical graph (Fig 4), and finally the conceptual model (Fig. 5) including the relationship cardinalities. Let us note that the figures do not exhibit the instances even if they are also transferred to the logical and conceptual levels by the reverse process. At the logical level, a unique ISA edge is generated since only the instance `'billy'` is defined with two different labels. This example illustrates the fact that our abstraction process is mainly based on labels attached to vertices and to edges. The labels define the semantics on which the reverse process groups together vertex instances as well as edge instances to constitute types which become entities and relationships at the conceptual level.

Building on our quality framework [18], we performed an evaluation process of the resulting conceptual model using two main criteria: expressiveness, and minimality. If the weight is one for entities and three for relationships as well as for ISA links, we obtain an expressiveness of 1 for the model at Fig. 5, since it contains all the concepts. Regarding minimality, the scores obtained by the model at Fig. 5 is equal to 1 since all redundancies are removed. The experiments on this case study (by adding more instances) led us to ascertain that correctness improves with the size of the physical graph. The reason is that having more instances facilitates the obtaining of correct cardinalities and the adequate positioning of properties and relationships in the model.

Our approach provides the Neo4j database user with 1) an automatic generation of the property graph, 2) an automatic generation of a conceptual representation. The reverse engineering process builds heavily on the node labels in order to elicit ISA relationships between categories. Moreover, it also uses the ISA relationship to avoid redundant relationships.

```
CREATE(shakespeare:Author{firstname:'William',lastname:'Sha
kespeare'}),(divorce:Play{title:'Divorcons'}),(victorien:Au
thor{firstname:'Victorien',lastname:'Sardou'}),(emile:Autho
r{firstname:'Emile';lastname:'Najac'}),(victorien)-
[:WROTE_PLAY{year:1880}]->(divorce),(victorien)-
[:WROTE_PLAY{year:1880}]->(divorce),(juliusCaesar:Play
{title:'Julius Caesar'}),(rsc:Company {name:'RSC'}),
(shakespeare)-[:WROTE_PLAY {year:1599}]->(juliusCaesar),
(theTempest:Play {title:'The Tempest'}),
(shakespeare)-[:WROTE_PLAY {year:1610}]->(theTempest),
 (production1:Production {name:'Julius Caesar'}),
 (production1)-[:PRODUCTION_OF]->(juliusCaesar),
(performance1:Performance {date:20120729}),
(performance1)-[:PERFORMANCE_OF]->(production1),
(production2:Production {name:'The Tempest'}),
 (production2)-[:PRODUCTION_OF]->(theTempest),
(performance2:Performance {date:20061121}),
(performance2)-[:PERFORMANCE_OF]->(production2),
(performance3:Performance {date:20120730}),
(performance3)-[:PERFORMANCE_OF]->(production1),
(billy:User :Author{id:'45'}),(rsc)-[:PRODUCED]-
>(production2),(review:Review {rating:5, review:'This was
awesome!'}),(billy)-[:WROTE_REVIEW]->(review),(review)-
[:RATED]->(performance1),(theatreRoyal:Venue {name:'Theatre
Royal'}),(performance1)-[:VENUE]->(theatreRoyal),
```

```
(performance2)-[:VENUE]->(theatreRoyal),(performance3)-
[:VENUE]->(theatreRoyal),(greyStreet:Street {name:'Grey
Street'}),(theatreRoyal)-[:STREET]->(greyStreet),
(newcastle:City {name:'Newcastle'}),(greyStreet)-[:CITY]-
>(newcastle),(tyneAndWear:County {name:'Tyne and Wear'}),
(newcastle)-[:COUNTY]->(tyneAndWear),(stratford)-
[:COUNTRY]>(england),(england:Country{name:'England'}),(tyn
eAndWear)-[:COUNTRY]->(england),(stratford:City
{name:'Stratford upon Avon'}),(rsc)-[:BASED_IN]-
>(stratford),(shakespeare)-[:BORN_IN]->stratford)
```

Fig. 3. Cypher code for the example (adapted from [17])

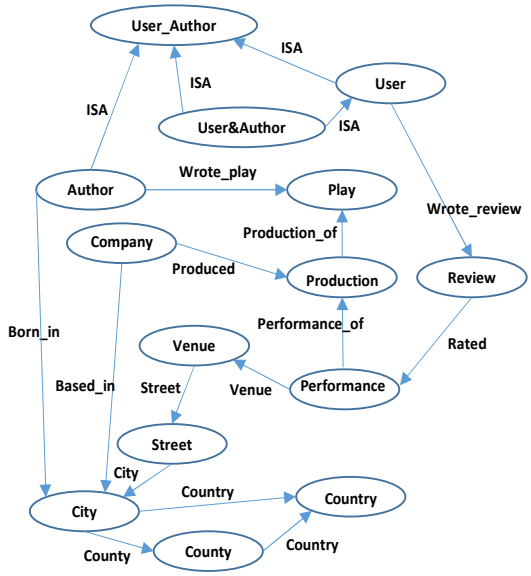

Fig. 4. Logical graph

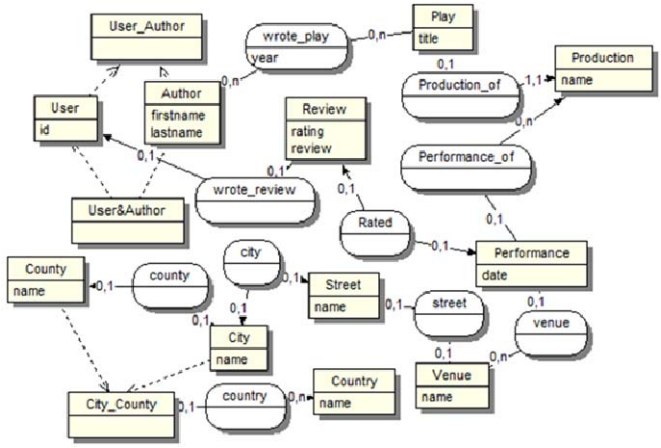

Fig. 5. Conceptual model resulting of the approach

V. Conclusions and Further Research

This paper presents a Model Driven Approach dedicated to reverse engineering of NoSQL property graph databases. It is applied to Neo4j graph databases. We present an illustrative scenario and evaluate the reverse engineering approach. The first results show that our approach is rather robust, reaches a high level of completeness taking advantage from data instances, and leads to a minimal schema since it discards semantic redundancies.

This work can be easily extended for reverse engineering of other NoSQL database systems, including key-value or document databases. Extending our physical and logical models to include hypergraphs could enable the generation of N-ary relationships. Our approach will benefit from more experiments on large databases. To this end, we are currently developing a prototype implementing the meta-models and the transformation rules.

References

[1] P. Atzeni, F. Bugiotti, L. Cabibbo, R. Torlone, "Data modeling in the NoSQL world", Computer Standards & Interfaces, 2016.

[2] H. Vera, M.H. Wagner Boaventura, M. Holanda, V. Guimaraes , F. Hondo, "Data Modeling for NoSQL Document Oriented Databases", in CEUR Workshop Proceeding, 2015, pp. 129-135.

[3] Y. Li, P. Gu, C. Zhang, "Transforming UML Class Diagrams into HBase Based on Meta-model", Int. Conf. on Information Science, Electronics & Electrical Eng., 720-724 (2014).

[4] S. Banerjee, A. Sarkar, "Logical Level Design of NoSQL Databases", IEEE Region 10 Conference (TENCON), 2016.

[5] T. Olier, "Database Design Using Key-Value Tables", (http://www.devshed.com/c/a/mysql/database-design-using-key-value-tables/), 2006.

[6] F. Di Tria, E. Lefons, F. Tangorra, "Design process for Big Data Warehouses", DSAA 2014, pp. 512-518.

[7] G. Daniel, G. Sunyé, J. Cabot, " UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases", In: 35th International Conference on Conceptual Modeling (ER2016), Japan.

[8] D. Aggarwal, K.C. Davis, "Employing Graph Databases as a Standardization Model towards Addressing Heterogeneity", 17th Int. Conf. on Information Reuse and Integration, IEEE, 2016.

[9] A. Jahangard-Rafsanjani, S.H. Mirian-Hosseinabadi, "A Model-Driven Approach to Semi-Structured Database Design", Frontiers of Computer Science 9(2), pp 237-252 (2015).

[10] S. Scherzinger, M. Klettke, U. Storl, "Managing Schema Evolution in NoSQL Data Stores". Proc. DBPL, CoRR, 2013.

[11] R. Yangui, A. Nabli, F. Gargouri, "Automatic Transformation of Data Warehouse Schema to NoSQL Database: Comparative Study", Procedia Computer Science 96 (2016), pp. 255-264.

[12] M. Chevalier, M. El Maliki, A. Kopliku, O. Teste, R. Tournier, "Implementation of Multidimensional Data Bases with Document-Oriented NoSQL", in 17th International Conference on Big Data Analytics and Knowledge Discovery, (Dawak'2015), Cham, (Switzerland), LNCS, vol. 9263, Springer, 2015. pp. 379-390.

[13] R.H. Chiang, T.M. Barron, V.C. Storey, "Reverse engineering of relational databases: Extraction of an EER model from a relational database". *Data & Knowledge Engineering,* 1994, *12*(2), pp. 107-142.

[14] N. Lammari, I. Comyn-Wattiau, J. Akoka, "Extracting generalization hierarchies from relational databases: A reverse engineering approach", Journal of Data and Knowledge Engineering, 2007, 63(2):568–589

[15] D.S. Ruiz, S.F. Morales, J.G. Molina, "Inferring Versioned Schemas from NoSQL Databases and its Applications", in 34th International Conference on Conceptual Modeling (ER 2015), 2015, pp. 467–480.

[16] K. Lamhaddab, K. Elbaamrani, "Model Driven Reverse Engineering: Graph Modeling for Mobiles Platforms", 15th International Conference on Intelligent Systems DeSign and Applications (ISDA), 2015.

[17] I. Robinson, J. Webber, E. Eifrem, "Graph databases: new opportunities for connected data", O'Reilly Media, Inc., 2015.

[18] S. S. S. Cherfi, J. Akoka, I. Comyn-Wattiau, "Conceptual modeling quality-from EER to UML schemas evaluation". In International Conference on Conceptual Modeling (pp. 414-428). Springer (2002).