A Rule-based Conversion of an EER Schema to Neo4j Schema Constraints

Telmo Henrique Valverde da Silva[1], **Ronaldo dos Santos Mello**[2]

[1]INE - Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC

[2]PPGCC - INE - Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC

telmo.trooper@gmail.com,r.mello@ufsc.br

Abstract. *Several application domains hold highly connected data, like supply chain and social network. In this context, NoSQL graph databases raise as a promising solution since relationships are first class citizens in their data model. Nevertheless, a traditional database design methodology initially defines a conceptual schema of the domain data, and the Enhanced Entity-Relationship (EER) model is a common tool. This paper presents a rule-based conversion process from an EER schema to Neo4j schema constraints, as Neo4j is the most representative NoSQL graph database management system with an expressive data model. Different from related work, our conversion process deals with all EER model concepts and generates rules for ensuring schema constraints through a set of Cypher instructions ready to run into a Neo4j database instance, as Neo4J is a schemaless system, and it is not possible to create a schema* $a\ priori$*. We also present an experimental evaluation that demonstrates the viability of our process in terms of performance.*

1. Introduction

NoSQL graph Databases (DBs) are becoming popular as they are suitable to application domains where relationships are first class citizens, like social network, telecommunication and supply chain [Robinson et al. 2015]. Different from traditional relational DBs, which impose integrity constraints over a predefined schema, and sometimes require a lot of additional tables to model relationships, NoSQL graph DBs provide more scalability and availability as they are usually *schemaless* and follow a *weak consistency* paradigm based on the BASE properties [Sadalage and Fowler 2012].

Nevertheless, schema constraints control for NoSQL graph DBs can be important, specially when a DB design methodology is considered [Elmasri and Navathe 2016]. An initial conceptual modeling step, for example, imposes several restrictions over a relationship type r_i, including valid entity types associated to r_i, cardinality constraints, and allowed attributes for r_i. In a supply chain scenario, for example, we may have specific routes that limit the road and city connections on which a mean of transportation must move. To be schema-aware is also important to several data management tasks, like data integration and optimization of query processing [Ruiz and et al. 2015].

This paper tries to cope with this problem of providing schema integrity control to NoSQL graph DBs by proposing a process that converts an *Extended Entity-Relationship (EER)* conceptual schema to a set of constraint specifications for the *Neo4j* DB Management System (DBMS)[1]. The EER model is a common tool for conceptual DB design,

[1]*https://neo4j.com*

and *Neo4j* is the most representative NoSQL graph DBMS[2]. We assume a *from-scratch* design of a Neo4j DB based on a conceptual modeling. Our main contributions are: *(i)* a set of conversion rules that maps EER schema constraints to *Cypher* instructions, and; *(ii)* a conversion process that considers the analysis of all EER model concepts. Cypher is the Neo4j data manipulation language[3].

Some efforts in the literature are similar to our proposal [Virgilio et al. 2014, Daniel et al. 2016, Akoka et al. 2017, Pokorny 2017, Sousa and Cura 2018]. However, they neither deal with all the EER concepts nor automatically generates a complete set of schema constraint specifications for a NoSQL graph DBMS language.

The rest of this paper is organized as follows. Section 2 formalizes the EER model and gives a background for Neo4j data model. Section 3 discusses related work and Section 4 details our conversion process. Section 5 evaluates our proposal and Section 6 is dedicated to the conclusion.

2. Background

2.1. EER Model

The EER model is a *de facto* standard for conceptual database design [Elmasri and Navathe 2016]. It extends the ER model, which comprises real-world entities, relationships, as well as attributes for both of them.

Besides the aforementioned concepts, an EER schema allows the definition of: *(i)* optional, multivalued and composite attributes; *(ii)* inheritance relationships (entities that inherit attributes and relationships from another entity); *(iii)* union types (entities unified into another entity that categorizes them); *(iv)* weak entities (an entity that depends on one or more entities to exist); *(v)* associative entities (a relationship promoted to an entity in order to be related to another entities); and *(vi)* n-ary relationships (a relationship connecting three or more entities). We next formalize an EER schema.

Definition 1 ***(EER Schema)****. An EER schema is a tuple* $eer = (n, E, R, SP, UT)$*, where* n *is the name of the schema,* E *is the set of entities,* $|E| > 0$*,* R *is the set of relationships,* $|R| >= 0$*,* SP *is the set of specialization relationships,* $|SP| >= 0$*, and* UT *is the set of union type relationships,* $|UT| >= 0$*;*

Definition 2 ***(Entity)****. An entity* $e_i \in eer.E$ *is a tuple* $e_i = (n, type, r_{base}, R_e, A_e, A_{id})$*, where* n *is the name of the entity,* $type \in \{regular, weak, associative\}$ *is the entity type,* $r_{base} \in eer.R$ *is the relationship promoted to an associative entity, if* $e_i.type = 'associative'$*,* $R_e \subseteq eer.R$ *is the set of relationships that connects* e_i *to other entities, if* $e_i.type = 'weak'$*,* A_e *is the set of* e_i *attributes,* $|A_e| > 0$*, and* A_{id} *is the set of attributes that solely identifies* e_i*,* $A_{id} \subseteq A_e$ *and* $|A_{id}| >= 0$*.*

In case of a weak entity, $e_i.R_e$ holds the relationships with the entities on which e_i depends for existing. Figure 1 shows an EER schema for a library domain. *Books*, *Copies* and *loan* are examples of a regular, weak and associative entity, respectively.

[2] *https://db-engines.com/en/ranking/graph+dbms*

[3] *https://neo4j.com/developer/cypher/*

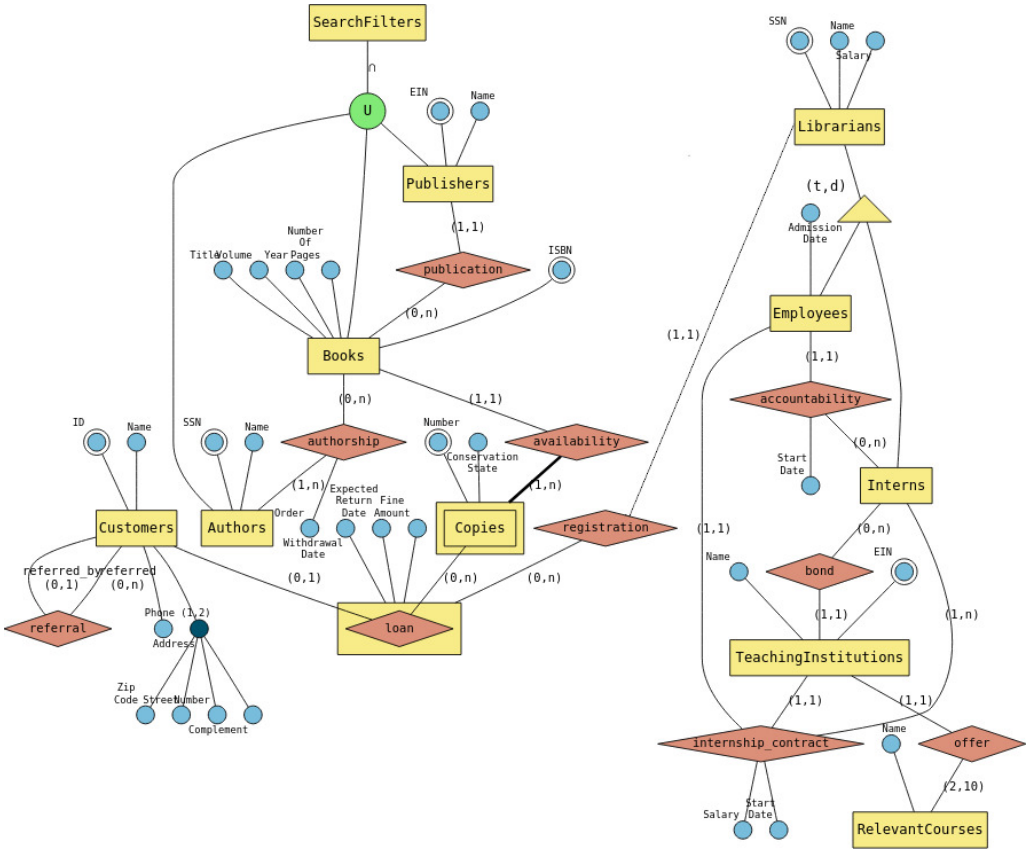

Figure 1. An EER schema example

Definition 3 ***(Relationship).*** *A relationship* $r_j \in eer.R$ *is a tuple* $r_j = (n, type, E_r, A_r, A_{id})$*, where* n *is the name of the relationship,* $type \in \{regular, recursive, n-ary\}$*,* $E_r = \{(e_1, c_1, r_1), ..., (e_n, c_n, r_n)\}$ *is the set of* r_j *connections, with* $(e_x, c_x, r_x) \in E_r$ *being an entity* $e_x \in eer.E$*,* $c_x = (c_{min}, c_{max})$ *the cardinality constraints for* e_x*,* r_x *is an optional role if* $r_j.type \neq' recursive'$*,* A_r *is the set of* r_j *attributes,* $|A_r| >= 0$*, and* A_{id} *is the set of attributes that helps in identifying* r_j*,* $A_{id} \subseteq A_r$ *and* $|A_{id}| >= 0$*.*

A regular relationship is a usual binary relationship connecting two different entities, and a recursive one has two connections with the same entity. A role in a relationship is a semantic meaning that can be defined for better sake of relationship understanding. Examples in Figure 1 are *referred_by* and *referred* in the recursive relationship *referral*. We also have examples of a 3-ary relationship (*internship_contract*) and a recursive relationship with roles (*referral*).

Definition 4 ***(Attribute).*** *An attribute* $a_k \in eer.E.A_e \cup eer.R.A_r$ *is a tuple* $a_k = (n, type, c_a, A_a)$*, where* n *is the attribute name,* $type \in \{regular, composite, multivalued, comp\_mv\}$*,* $c_a = (c_{min}, c_{max})$ *is the attribute cardinality, and* A_a *is the set of inner attributes if* a_k *is a composite attribute.*

An attribute with type $comp\_mv$ is an attribute that is composite and multivalued. The attribute cardinality allows the definition of optional, regular or multivalued attribute, like *(0,1)*, *(1,1)* or *(1,n)*, respectively. *Address* and *Phone* are examples of composite and multivalued attributes in Figure 1, respectively.

Definition 5 ***(Specialization).*** *A specialization* $s_l \in eer.SP$ *is a tuple* $s_l = (e_{super}, E_{sub}, type)$*, where* $e_{super} \in eer.E$ *is the generic entity,* $E_{sub} \subset eer.E$ *is the set of specialized entities, and* $type$ *is the type of the specialization relationship, with* $type \in \{(t,d),(t,s),(p,d),(p,s)\}$*;*

Definition 6 ***(Union Type).*** *A union type* $u_m \in eer.UT$ *is a tuple* $u_m = (category, E_{members})$*, where* $category \in eer.E$ *is the entity that unifies a set of entities into a category, and* $E_{members} \subset eer.E$ *is the set of entities that belongs to the category.*

The types of specialization relationships are a combination of two possible settings: *total* or *partial*, as well as *disjoint* or *shared*. The hierarchy headed by *Librarians* is an example of a total and disjoint specialization in Figure 1. We also have a union type that defines a category *SearchFilters* comprising *Authors*, *Books* and *Publishers*.

2.2. Neo4J Data Model

A NoSQL graph DB is usually a *property graph* of data, *i.e.*, a graph structure where vertices may hold properties [Angles 2012, Sadalage and Fowler 2012, Robinson et al. 2015]. Despite this consensus, there is not a standard graph data model. *Vertex* and *edge* are basic concepts, as well as vertex *properties*. A vertex models a *labeled* real-world object, an edge represent a relationship between two vertexes, and a property is an attribute-value pair where the value is restricted to a datatype. Most of the data models also provide additional details for edges: *properties, label* and *relationship direction*.

A few data models support *multiple labels* for vertexes, in sense that an object belongs to more than a type (*e.g., customer* and *employee*). It is also possible that properties hold *complex datatypes* (*e.g., arrays, geographic types*) besides traditional atomic ones.

The Neo4j DBMS supports all of these aforementioned data model concepts. Figure 2 shows an example of a Neo4j DB that is in accordance to the EER schema of Figure 1. The vertexes exhibit the name of one of their labels, and the same holds for the edges.

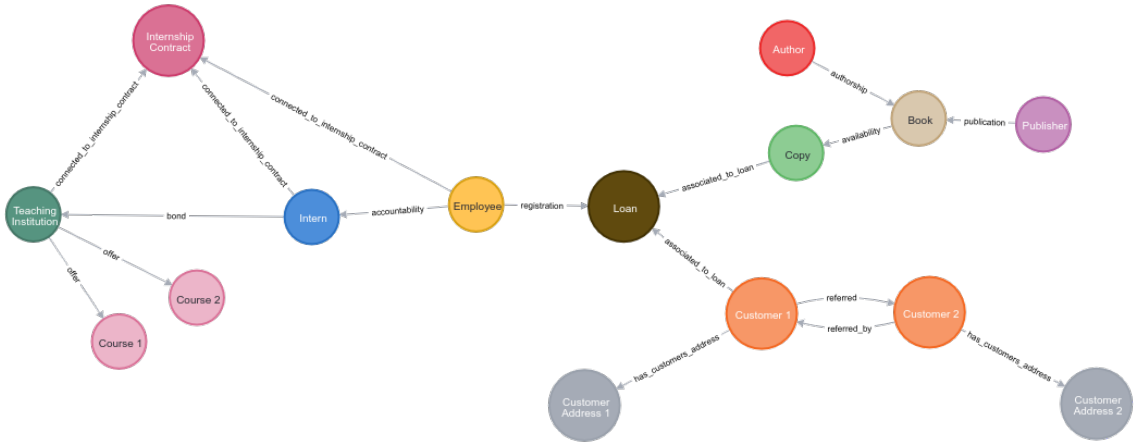

Figure 2. A Neo4J database

3. Related Work

Some approaches in the literature are similar to our work. *UMLtoGraphDB* proposes a mapping of a *UML* conceptual schema, with embedded *OCL* business rules, to a graph

DB schema described in the *Gremlin* language [Daniel et al. 2016]. Such a schema can be then extended by the designer to generate schema constraints in a NoSQL graph DBMS.

The work of Akoka et al. [Akoka et al. 2017] introduces a limited EER metamodel with some statistics about entities, relationships, attributes and specializations volume and variety in order to guide the designer to choice the best mapping alternatives to a property graph schema. In fact, it is a theoretical conversion solution, having not focus on the generation of schema constraints to one or more NoSQL graph DBMS.

The *Binary ER* approach maps an ER schema to a limited set of corresponding graph schema constraints for Neo4j, Titan and OrientDB graph DBMSs [Pokorny 2017]. Another work called *Orient ER* also converts a basic ER schema to a labeled graph [Virgilio et al. 2014]. However, it does not generate schema constraints for a NoSQL graph DBMS. Instead, it only provides a set of rules to be considered by the designer when creating a graph DB. Finally, we have a work that proposes a set of extensions to the Cypher language to specify DB schema constraints [Sousa and Cura 2018]. The focus is on constraint specifications to guarantee disjoint, uniqueness and identification control.

Different from the related work, our approach treats all EER concepts and proposes an automatic process to generate a complete set of schema control instructions for Neo4j from an EER schema, being a more robust solution for this NoSQL graph DBMS.

4. Proposed Approach

4.1. Conversion Strategy

Our strategy to force Neo4j data to respect a conceptual schema is to specify a set of *Cypher* constraints. Two Cypher instructions allow data constraint specification:

- *Constraint*: it has the format `CREATE CONSTRAINT ON` *label* `ASSERT` *rule*, where *label* is a vertex or edge label, and *rule* is one of the following constraints:
  - `EXISTS(`*property*`)`: mandatory property;
  - *property* `IS UNIQUE`: uniqueness constraint for a property;
  - *(*$property_1$*, ...,* $property_n$*)* `IS NODE KEY`: node composite identifier.
- *Trigger*: it has the format `MATCH` *pattern* [`WHERE` *condition*] *action*, where *pattern* is the path expression to be matched into the DB graph, *condition* is an optional set of filters that must be true in order to fire the action, and *action* is the set of Cypher instructions to be executed.

We consider these instructions in the definition of *schema constraint rules*, which are the core of our conversion process. Figure 3 shows examples of constraint definitions in Cypher. Figure 3 (a) shows a trigger that controls valid labels for vertexes. This is one of the generated triggers for the EER schema of Figure 1.

The same holds for relationship control, as shown in Figure 3 (b). It limits the valid relationship types to the ones defined in the EER schema. Figure 3 (c), in turn, shows constraint definitions that control valid properties for an entity or relationship. It defines constraints over the *Publishers*' attributes. Finally, Figure 3 (d) illustrates a trigger that controls valid relationship cardinalities. It avoids that a *Copy* be associated to more than one *Book* (see Figure 1). If it happens, the extra connections of a copy are deleted.

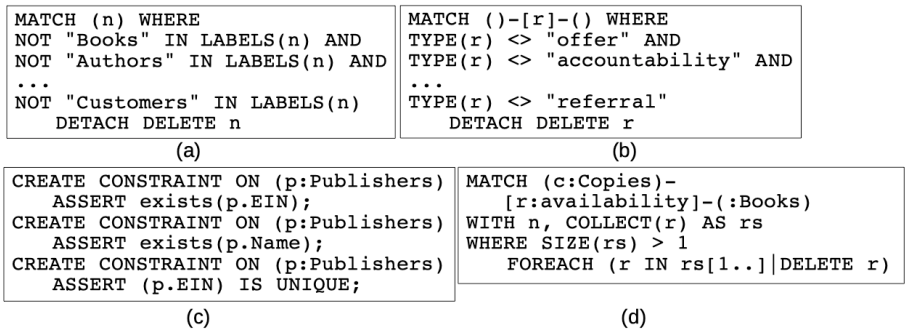

Figure 3. Examples of constraint definitions in Cypher

Our conversion strategy is summarized in Table 1. We base some mappings from the related work, and define several other ones to deal with EER concepts not treated before. From all related approaches we borrow the consensual mapping of regular entities, attributes and relationships, and the n-ary relationship conversion reasoning comes from *UMLtoGraphDB* [Daniel et al. 2016]. Two works [Akoka et al. 2017, Pokorny 2017] propose conversions for specializations by generating IS-A edges between the vertexes that represent generic and specialized entites. Instead, we propose multilabels for the vertexes of specialized entities to avoid such edges. There is only one work that also introduces a conversion algorithm [Sousa and Cura 2018]. However, it covers only regular entities, relationships and attributes.

In short, we map entities to vertexes as they represent relevant real-world facts. It includes entities in hierarchies (specializations and union types). In turn, relationships are mapped to edges as they represent entities' associations. The exception is a n-ary relationship, which becomes a vertex v_{n_ary} as it connects more than two entities, being not possible to be represented as an edge due to its limitation to connect only two facts. In this case, v_{n_ary} has edges to all vertexes representing the associated entity occurrences.

Table 1. EER-to-Neo4j conversion strategy

EER	Neo4j	EER	Neo4j
Regular entity	vertex	Regular relationship	edge
Weak entity	vertex	Recursive relationship	edges
Associative entity	vertex	N-ary relationship	vertex
Regular attribute	property	Specialization	vertexes
Composite attribute	vertex	Union type	vertexes
Multivalued attribute	array property		

Entity and relationship attributes are converted to vertex and edge properties, respectively. The exception is a composite attribute, which is mapped to a vertex that holds the inner attributes, as Neo4j does not support composite properties. On the other hand, Neo4j supports properties with an array domain, which is suitable for multivalued attributes. More details are given by the conversion rules formalized in the next section.

4.2. Conversion Rules

The conversion rules are the core of our conversion process. Definitions 7 and 8 show two basic rules responsible to specify valid *labels* and *property names* for vertexes and edges

that correspond to entities and relationships, respectively. By label we mean the name of a entity, relationship or composite attribute. The first one contributes to the definition of a trigger that control valid labels for vertexes and edges, as exemplified in Figure 3 (a) and Figure 3 (b). The second one generates constraints for controlling valid vertex and edge properties. Figure 3 (c) exemplifies these constraints for the *Publishers* vertex. The first part of this rule also considers composite attributes, which hold properties and, as a consequence, are converted to vertexes. More details are given in Section 4.3.

Definition 7 *(**Label Rule: Label(label, type)**). include a predicate* `NOT "label" IN LABELS(n)` *in Valid_Vertex_Labels trigger, if* $type = 'vertex'$*; include a predicate* `TYPE(r) <> "label"` *in Valid_Edge_Labels trigger, if* $type = 'edge'$*;*

Definition 8 *(**Property Rule: Prop(property, concept)**). generate a constraint:*
(i) `CREATE CONSTRAINT ON (c:concept.n) ASSERT EXISTS (c.property)`*, if* $concept \in eer.E$ *or* $concept \in eer.E.A_e \cup eer.R.A_r$*;*
(ii) `CREATE CONSTRAINT ON ()-[r:concept.n]-() ASSERT EXISTS (r.property)`*, if* $concept \in eer.R$*.*

Definition 9 controls valid identifications for vertexes and edges based on the identifiers of entities and relationships. If an identifier is composed of one attribute, then a `UNIQUE` constraint is created. Otherwise, a `IS NODE KEY` constraint is created. Definition 10 rules the conversion of a multivalued attribute through the creation of a trigger that avoids an array property to disrespect the minimal and maximal number of values.

Definition 9 *(**Identity Rule: Id(property_set, concept)**). generate a constraint:*
(i) `CREATE CONSTRAINT ON (c:concept) ASSERT (c.`a_u`.n) IS UNIQUE`*, if* $|property\_set| = 1$ *and property_set =* $\{a_u\}$*;*
(ii) `CREATE CONSTRAINT ON (c:concept) ASSERT (`$a_1.n, ..., a_x.n$`) IS NODE KEY`*, if* $|property\_set| > 1$ *and property_set =* $\{a_1, ..., a_x\}$*;*

Definition 10 *(**Multivalued Attribute Rule: MV_Att(**a_i**,** c_{name}**)**). generate a trigger* `MATCH (n:"`c_{name}`") WHERE size(n."`$a_i.n$`") <` $a_i.c_a.c_{min}$ `OR size(n."`$a_i.n$`") >` $a_i.c_a.c_{max}$ `DETACH DELETE n`*.*

Valid weak entities are ruled by Definition 11. This rule avoids a vertex that correspond to a weak entity we to exist without being connected to the vertexes that represent the entities on which we depends on (*strong entities*). It also extends we identification to add the identification of its strong entities, as a identification dependency also holds.

Definition 11 *(**Weak Entity Rule: WEnt(**we**)**). for each entity* $e_s \in we.R_e.E_r$*,* $e_s \neq we$*:*
(i) generate a trigger `MATCH (n:`$we.n$`) WHERE NOT (:`$e_s.n$`)-[:has]-(n) DETACH DELETE n;`
(ii) $we.A_{id} \leftarrow we.A_{id} \cup e_s.A_{id}$*.*

Definition 12 controls valid connections between vertexes according to the cardinalities of a relationship r involving a source entity *e1* and a target entity *e2*. Part *(i)* avoids more than one connection if the maximal cardinality (max_{card}) is 1. Part *(ii)* avoids a number of connections less than the minimal cardinality, and part *(iii)* avoids a number of connections higher than max_{card}, if max_{card} is fixed. Definition 13, in turn, rules a recursive relationship r_j: two edges are necessary for the two r_j roles connecting two vertex instances that represent the same entity e_i that holds r_j. If one of the two edges is missing or they are not connected to e_i vertexes, then the edges must be deleted.

Definition 12 ***(Cardinality Rule: Card(***$r.n$***,*** $e1.n$***,*** $e2.n$***,*** min_{card}***,*** max_{card}***)). generate the triggers:***
(i) `MATCH (n:"`$e1.n$`")-[r:"`$r.n$`"]-(:"`$e2.n$`") WITH n, COLLECT(r) AS rs WHERE SIZE(rs) > 1 FOREACH (r IN rs[1..] | DELETE r),` ***if*** $max_{card} = 1$***;***
(ii) `MATCH (n:"`$e1.n$`")-[r:"`$r.n$`"]-(:"`$e2.n$`") WITH n, COLLECT(r) AS rs WHERE SIZE(rs) <` min_{card} `FOREACH (r IN rs | DELETE r),` ***if*** $min_{card} > 1$***;***
(iii) `MATCH (n:"`$e1.n$`")-[r:"`$r.n$`"]-(:"`$e2.n$`") WITH n, COLLECT(r) AS rs WHERE SIZE(rs) >` max_{card} `FOREACH (r IN rs[`max_{card}`..] | DELETE r),` ***if*** $max_{card} \neq$ ***"N";***

Definition 13 ***(Recursive Rule: Recur(***r_j***)). generate the triggers:***
(i) `MATCH (n)-[r:"`$r_j.E_r.r_1$`"]->() WHERE NOT "`$r_j.E_r.e_1.n$`" IN LABELS(n) AND NOT ()-[r:"`$r_j.E_r.r_2$`"]->(n) DELETE` r_j***;***
(ii) `MATCH (n)-[r:"`$r_j.E_r.r_2$`"]->() WHERE NOT "`$r_j.E_r.e_1.n$`" IN LABELS(n) AND NOT ()-[r:"`$r_j.E_r.r_1$`"]->(n) DELETE` r_j***.***

The last two rules control valid entities' hierarchies. The first one (Definition 14) regards specialization hierarchies, and the second one (Definition 15) regards union type hierarchies. The $Spec(sp)$ rule initially requires that all vertexes that correspond to specialized entities hold a label for the generic entity besides its own label. Part *(ii)* controls disjoint specializations. It avoids that a vertex holds a label for more than one specialized entity. Part *(iii)* rules total specializations by prohibiting a vertex with a generic entity label that does not also hold a label for any of the specialized entities. Partial and shared specializations are more flexible, so no constraints are needed for them.

Definition 14 ***(Specialization Rule: Spec(***sp***)). generate the triggers:***
(i) `MATCH (n) WHERE ("`$sp.E_{sub}.e_1.n$`" IN LABELS(n) OR ... OR "`$sp.E_{sub}.e_n.n$`" IN LABELS(n)) AND NOT "`$sp.e_{super}.n$`" IN LABELS(n) DETACH DELETE n;`
(ii) `MATCH (n) WHERE ("`$sp.E_{sub}.e_1.n$`" IN LABELS(n) AND "`$sp.E_{sub}.e_2.n_e$`") OR ... OR ("`$sp.E_{sub}.e_1.n$`" IN LABELS(n) AND ... AND "`$sp.E_{sub}.e_n.n$`") IN LABELS(n) DETACH DELETE n,` ***if*** $sp.type = (t, d)$ ***or*** $sp.type = (p, d)$***;***
(iii) `MATCH (n:`$sp.e_{super}.n$`) WHERE NOT "`$sp.E_{sub}.e_1.n$`" IN LABELS(n) AND ... AND NOT "`$sp.E_{sub}.e_n.n$`" IN LABELS(n) DETACH DELETE n,` ***if*** $sp.type = (t, d)$ ***or*** $sp.type = (t, s)$***;***

Definition 15 ***(Union Type Rule: Union(***ut***)). generate the triggers:***
(i) `MATCH (n:`$ut.category.n$`) WHERE NOT "`$ut.E_{members}.e_1.n$`" IN LABELS(n) AND ... AND NOT "`$ut.E_{members}.e_n.n$`" IN LABELS(n) DETACH DELETE n;`
(ii) `MATCH (n) WHERE NOT n:`$ut.category.n$ `AND (n:`$ut.E_{members}.e_1.n$ `OR ... OR n:`$ut.E_{members}.e_n.n$`) DETACH DELETE n.`

The $Union(ut)$ rule requires that the label for the vertex v_i that represent the category never be alone, *i.e.*, at least one label of a member entity must also be present at

v_i. The opposite situation is also required: a vertex with a label of a member entity must also holds the label of the entity that represents the category.

4.3. Conversion Process

Our conversion process adapts the methodology for logical design of relational DBs [Elmasri and Navathe 2016] to NoSQL graph DBs. The input is an EER schema and the output is a set of Neo4j constraints. It has four steps: *(i) Label Constraints Initialization; (ii) Entity Conversion; (iii) Hierarchy Conversion;* and *(iv) Relationship Conversion*.

The first step is a pre-processing procedure that initializes two triggers related to valid labels for vertexes and edges that correspond to entities and relationships. In short, we generate the headers "`MATCH (n) WHERE`" and "`MATCH ()-[r]-() WHERE`", which are extended with valid labels for entities and valid types for relationships while the EER entities and relationships are analyzed, respectively. Examples of these triggers are shown in Figure 3 (a) and Figure 3 (b).

The next step converts the entities and their attributes in order to constraint most of the valid vertexes. Then, we proceed the conversion of the hierarchies to specify additional vertex constraints that control valid hierarchical relationships (see Definition 13 and Definition 14) followed by the conversion of the relationships, which constraints most of the valid edges. The overall process is detailed in Algorithm 1.

The process first initializes the set of output Cypher constraints C_{cons} (line 2) as an empty set, and executes the first step (line 3). In the following, we deal with entity conversion (lines 4 to 9). We execute the $Label$ rule to include valid vertexes' labels in the $Valid\_Vertex\_Labels$ trigger (line 4), and then we analyze again all entities (line 5) to generate constraints for weak entities (line 6) and entity identification (line 8).

We define another algorithm for attribute conversion (Algorithm 2). It receives an entity, relationship or composite attribute, its set of attributes, and C_{cons}, and appends specific constraints for attributes into C_{cons}. The algorithm first verifies if an attribute a_k is a composite or composite/multivalued one (line 3). If so, a_k generates a vertex v_{ak} (line 4) and an edge between the vertex that represent the concept c_i and v_{ak} (line 5). The vertex label is a concatenation of c_i and a_k names, and the edge label includes additionally the string *'has'* as a prefix. One example is shown in Figure 2 for the conversion of the composite attribute $Address$ in Figure 1. Next, the a_k nested attributes are recursively converted (line 6). We also deal with the cardinality constraints of the relationship between c_i and v_{ak} (an specific procedure - line 7). In this case, the cardinality in the $c_i \rightarrow v_{ak}$ direction is the a_k cardinality, and the opposite direction is *(1,1)* as v_{ak} is dependent on c_i. Finally, if a_k is not a composite one (line 9), we consider it as a valid property for c_i (line 10), and, at last, we deal with the conversion of a multivalued a_k (line 11). For sake of paper space, we do not detail the case where c_i is a recursive relationship. In this situation, $c_i.n$ is replaced by the c_i roles' names, generating double invocations for $Label$, $Card$ and $Prop$ rules to constrain each one of the two roles names, cardinalities and properties, respectively.

Back to Algorithm 1, we deal with the conversion of EER hierarchies (lines 10 and 11), and then relationships (lines 12 to 48). We first check if a relationship r_j is a n-ary one or r_j holds a composite attribute. In these cases, r_j must be converted to a vertex as it has connections with several entities, or it must be connected to another vertex representing

Algorithm 1: EER-to-Neo4j conversion

Input : EER schema eer
Output: set of Cypher constraint instructions C_{cons}

1 **begin**
2 $C_{cons} \leftarrow \emptyset$;
3 $Init(Valid\_Vertex\_Labels, Valid\_Edge\_Labels)$;
4 **for** *each* $e_i \in eer.E$ **do** $Label(e_i.n, 'vertex')$;
5 **for** *each* $e_i \in eer.E$ **do**
6 **if** $e_i.type = 'weak'$ **then** $C_{cons} \leftarrow C_{cons} \cup WEnt(e_i)$;
7 $Attributes\_Conversion(e_i, C_{cons})$;
8 **if** $|e_i.A_{id}| > 0$ **then** $C_{cons} \leftarrow C_{cons} \cup Id(e_i.A_{id}, e_i.n)$
9 **end**
10 **for** *each* $s_l \in eer.SP$ **do** $C_{cons} \leftarrow C_{cons} \cup Spec(s_l)$;
11 **for** *each* $u_m \in eer.UT$ **do** $C_{cons} \leftarrow C_{cons} \cup Union(u_m)$;
12 **for** *each* $r_j \in eer.R$ **do**
13 **if** *exists* $a_k \in r_j.A_r (a_k.type = 'composite'$ *or* $a_k.type = 'comp\_mv')$ *or* $r_j.type = 'n-ary'$ **then**
14 $Rel\_to\_Vertex(r_j, C_{cons})$
15 **else**
16 **if** $r_j = e_i.r_{base} \mid e_i \in eer.E$ *and* $e_i.type = 'associative'$ **then**
17 **if** $r_j.type = 'recursive'$ **then**
18 $Cardinalities(e_i.r_{base}, 'recursive', C_{cons})$;
19 $Recur(r_j)$
20 **else**
21 $Cardinalities(e_i.r_{base}, 'associative', C_{cons})$
22 **end**
23 **if** $|r_j.A_{id}| > 0$ **then**
24 $r_j.A_{id} \leftarrow r_j.A_{id} \cup r_j.E_r.e_1.A_{id} \cup r_j.E_r.e_2.A_{id}$;
25 $C_{cons} \leftarrow C_{cons} \cup Id(r_j.A_{id}, e_i.n)$
26 **end**
27 **else**
28 **if** $r_j.type = 'recursive'$ **then**
29 $Label(r_j.E_r.r_1, 'edge'); Label(r_j.E_r.r_2, 'edge')$;
30 $Cardinalities(r_j, 'recursive', C_{cons})$;
31 $Recur(r_j)$
32 **else**
33 $Label(r_j.n, 'edge')$;
34 $Cardinalities(r_j, 'regular', C_{cons})$
35 **end**
36 $Attributes\_Conversion(r_j, C_{cons})$;
37 **if** $|r_j.A_{id}| > 0$ **then**
38 $r_j.A_{id} \leftarrow r_j.A_{id} \cup r_j.E_r.e_1.A_{id} \cup r_j.E_r.e_2.A_{id}$;
39 $C_{cons} \leftarrow C_{cons} \cup Id(r_j.A_{id}, r_j.n)$
40 **end**
41 **end**
42 **end**
43 **end**
44 $C_{cons} \leftarrow C_{cons} \cup Valid\_Vertex\_Labels \cup Valid\_Edge\_Labels$;
45 **return** C_{cons}
46 **end**

the composite attribute (lines 13 and 14). One example is the *Internship_Contract* vertex in Figure 2, which is the result of the conversion of the same name n-ary relationship in Figure 1. The Rel_to_Vertex procedure is responsible to the conversion of r_j. For sake of paper space, we do not detail it. In short, it defines constraints considering r_j a regular entity, and also constraints the allowed edges from r_j to the vertexes representing the associated entities or the vertex representing the composite attribute. These edges hold a predefined label with the format "$connected\_to\_$"$vertex\ label$"$\_r_j.n$" in the $vertex \rightarrow r_j$ direction (see Figure 2), and "$connected\_to\_r_j.n\_$"$vertex\ label$" in the opposite direction.

Finally, if r_j does not fit into the previous cases, then it is a regular or recursive relationship (lines 15 to 42), and the rules for valid relationship label, properties, cardi-

Algorithm 2: Attributes_Conversion

```
   Input : EER concept c_i, EER concept attribute set A_c, set of Cypher constraint instructions C_cons
1  begin
2      for each a_k ∈ A_c do
3          if a_k.type = 'composite' or a_k.type = 'comp_mv' then
4              Label(c_i.n + '_' + a_k.n, 'vertex');
5              Label('has' + '_' + c_i.n + '_' + a_k.n, 'edge');
6              Attributes_Conversion(a_k, a_k.A_a, C_cons);
7              Cardinalities(a_k,' attribute', C_cons)
8          else
9              C_cons ← C_cons ∪ Prop(a_k.n,c_i);
10             if a_k.type = 'multivalued' then C_cons ← C_cons ∪ MV_Att(a_k, c_i.n);
11         end
12     end
13 end
```

nalities, identification, and the recursive case are applied on it. At this point we still have to check if r_j is promoted to an associative entity (line 16) in order to provide the proper cardinality constraints with the involved inner entities (lines 17 to 26). In this case, the $Cardinalities$ procedure deals with the conversion of all cardinalities (line 21) related to the "$associated\_to\_$"$vertex$ edges (see Figure 2). The algorithm also verifies if r_j type is *recursive* (line 17) in order to constraint the r_j labels and cardinalities for the defined roles in both directions (see the relationships between $Customer1$ and $Customer2$ in Figure 2). Lines 27 to 41 treat an r_j that is not an associative entity, so a label constraint must be defined and its attributes converted. This conversion depends again on whether r_j is a recursive relationship or not (lines 28 to 35). At the end of Algorithm 1, we append the completed label triggers to C_{cons} and returns it. The worst-case complexity of this algorithm is *O(#E+#R)* as each entity and relationship type is accessed once.

5. Evaluation

Our initial intention was to evaluate the overhead introduced with the schema constraint checking when manipulation operations were executed over a Neo4j DB designed by our approach. However, Neo4j provides no immediate verification of triggers and constraints, which makes hard the overhead analysis. Thus, we decided to analyze the impact of executing our process over several sizes of EER schemas to evaluate its performance. We implemented a prototype for our conversion process, and a synthetic EER schema generator where we can set the number of created regular entities (E) and regular relationships (R), as well as the cardinality distribution for the relationships ((1,1), (1,n) and (m,n)), and the number of regular attributes. Four test scenarios were produced, starting from 50 E and 50 R, and doubling these numbers three times. For all of these scenarios we assume Es and Rs with five attributes, and an equal distribution of R cardinalities.

The tests were run in a computer with an Intel i7 3770 processor with 3.9 GHz and 16 Gb RAM. We performed twenty executions of each test scenario and got the average execution. Figure 4 shows the plotted times for the scenarios. Except for the 50/50-100/100 interval, we see a *linear* behaviour of our conversion process as the number or E and R doubles. It demonstrates the expected complexity of Algorithm 1.

6. Conclusion

This paper presents a rule-based conversion of an EER schema to Neo4j constraint specifications. These specifications allow Neo4j to accept only DB transactions that respect the

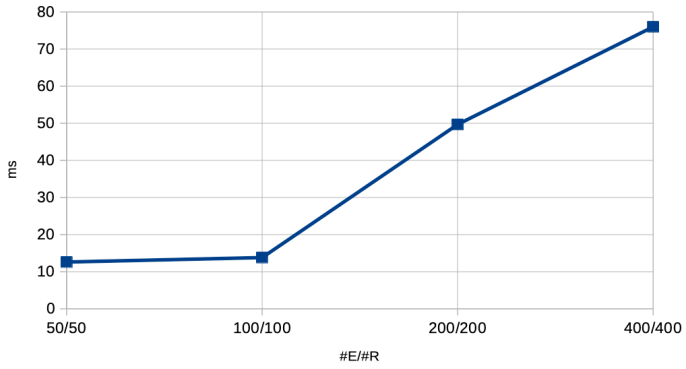

Figure 4. Processing times for the four test scenarios

EER schema, contributing to turn Neo4j a *schema-aware* DBMS. We also contribute to the design of a Neo4j DB from a conceptual schema. Our conversion algorithm presents a linear complexity w.r.t. to the number of EER entities and relationships, which makes it feasible to be reproduced and applied in real-world scenarios. Our proposal is also the only one that supports all EER concepts conversion by a complete automatized process.

As future works, we will try to have access to the source code of the related works to compare result quality and performance. We do not have success to obtain such a codes by now. We also intend to develop a Web tool to support our conversion process.

References

Akoka, J., Comyn-Wattiau, I., and Prat, N. (2017). A four v's design approach of nosql graph databases. In *ER Conference*, pages 58–68. Springer.

Angles, R. (2012). A Comparison of Current Graph Database Models. In *ICDE Conference Workshops*, pages 171–177. IEEE.

Daniel, G., Sunyé, G., and Cabot, J. (2016). Umltographdb: mapping conceptual schemas to graph databases. In *ER Conference*, pages 430–444. Springer.

Elmasri, R. and Navathe, S. B. (2016). *Fundamentals of Database Systems*. Pearson Higher Education, 7 edition.

Pokorny, J. (2017). Modelling of graph databases. *Journal of Advanced Engineering and Computation*, 1(1):04–17.

Robinson, I., Webber, J., and Eifrem, E. (2015). *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, Inc., 2 edition.

Ruiz, D. S. and et al. (2015). Inferring Versioned Schemas from NoSQL Databases and its Applications. *LNCS*, 9381:467–480.

Sadalage, P. J. and Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education.

Sousa, V. M. d. and Cura, L. M. d. V. (2018). Logical design of graph databases from an entity-relationship conceptual model. In *iiWAS Conference*, pages 183–189. ACM.

Virgilio, R. D., Maccioni, A., and Torlone, R. (2014). Model-driven design of graph databases. In *ER Conference*, pages 172–185. Springer.