Exploring the Visualization of Schemas for Aggregate-Oriented NoSQL Databases*

Alberto Hernández Chillón, Severino Feliciano Morales, Diego Sevilla Ruiz, and Jesús García Molina

Faculty of Computer Science, University of Murcia
Campus Espinardo, Murcia, Spain
{alberto.hernandez1,severino.feliciano,dsevilla,jmolina}@um.es

Abstract. The lack of an explicit data schema (*schemaless*) is one of the most attractive NoSQL database features for developers. Being schemaless, these databases provide a greater flexibility, as data with different structure can be stored for the same entity type, which in turn eases data evolution. This flexibility, however, should not be obtained at the expense of losing the benefits provided by having schemas: When writing code that deals with NoSQL databases, developers need to keep in mind at any moment some kind of schema. Also, database tools usually require the knowledge of a schema to implement their functionality. Several approaches to infer an implicit schema from NoSQL data have been proposed recently, and some utilities that take advantage of inferred schemas are emerging. In this article we focus on the requisites for the visualization of schemas for aggregate-oriented NoSQL Databases. Schema diagrams have proven useful in designing and understanding databases. Plenty of tools are available to visualize relational schemas, but the visualization of NoSQL schemas (and the variation that they allow) is still in an immature state, and a great R&D effort is required to achieve tools with the desired capabilities. Here, we study the main challenges to be addressed, and propose some visual representations. Moreover, we outline the desired features to be supported by visualization tools.

Keywords: NoSQL Databases, NoSQL schema, Schema Visualization

1 Introduction

The NoSQL term (*Not SQL/Not only SQL*) is used to denote a new generation of database systems that overcomes the limitations of relational systems to satisfy the requirements demanded by modern applications. A large number of companies have already embraced NoSQL databases, and this adoption will rise considerably in next years, as reported in [1,11]. Actually, the NoSQL

* Work partially supported by the Cátedra SAES of the University of Murcia (http://www.catedrasaes.org), a research lab sponsored by the SAES company (http://www.electronica-submarina.com/).

In: C. Cabanillas, S. España, S. Farshidi (eds.):
Proceedings of the ER Forum 2017 and the ER 2017 Demo track,
Valencia, Spain, November 6th-9th, 2017,
published at http://ceur-ws.org

term refers to a varied set of data modeling paradigms, divided on the following major categories: document, wide column, key-value stores and graph-based databases. While object aggregation is the main construct to represent data in the three former paradigms, graph-based models aim to represent connections (i.e. references) between objects [14].

The absence of an explicit schema (*schemaless* approach) is a convenient feature in most NoSQL systems, because it provides the required flexibility when the data structure varies often. Being schemaless, the database can store data with different structure (i.e. schema) for the same entity type (non-uniform data), and data evolution is favoured due to the lack of restrictions imposed on the data structure. However, removing the need of declaring explicit schemas does not have to be confused with the absence of a schema, as a schema is implicit into stored data and in the application code that deals with the database. The developers must always keep in mind the schema when they write or maintain code that is connected to the database. This is an error-prone task, more so when entities have a degree of variability. Moreover, some NoSQL database tools and utilities need to know the schema to offer even basic functionality. Therefore, the flexibility gained by being schemaless should not be at expense of the benefits provided by having explicit schemas. A growing interest in managing implicit NoSQL schemas is therefore arising, as evidenced in the schema inference approaches recently proposed [9,15,17]. Also, some existing relational modeling tools [4,3,5] are being extended to offer NoSQL modeling capabilities such as visualizing discovered schemas or database design.

The recent Dataversity report [1] has remarked that data modeling will be a crucial activity for NoSQL databases and has also drawn attention on the need for NoSQL tools that provide functionality similar to those available for relational databases. The authors of this report have identified schema visualization as an essential capability to be offered by NoSQL modeling tools.

Visualizing database schemas in form of diagrams is useful both to designers and developers. Designers can express the database structure at a high level of abstraction and better reason about efficiency and physical layout, and developers can write better code if they have a model that properly represents the database schema. Moreover, schema diagrams provide very useful documentation that facilitates the understanding of an existing database in performing evolution tasks. In fact, plenty of commercial and open-source modeling tools exist for relational databases.

While a relational database schema is formed by a set of entities and the relationships between them, in NoSQL databases several schemas can exist for the same entity. We refer to this feature as *versioned schemas*. This complicates the visualization of the global schema that includes all the entity version schemas and relationships between them. This issue is normally tackled by (i) showing entities whose schema is the union of all of its schemas, and (ii) ignoring the reference relationships between entities [4,3].

This article is focused on the visualization of NoSQL schemas, more specifically, we address the challenges that visualizing versioned schemas raise in the

case of aggregate-oriented NoSQL systems. For this, we will work on schemas inferred by applying the model-driven reverse engineering approach described in [15]. This schema inference strategy differs from other proposed approaches in three main characteristics: (i) It discovers all the schemas that the database stores for each entity, (ii) in addition to aggregation relationships, the reference relationships among entities are also discovered, and (iii) it represents inferred schemas in models that conform to an Ecore meta-model. We will define some possible schemas to be considered for NoSQL databases and we will discuss how they could be visually represented. We will also show the main design decisions to be addressed in the implementation of NoSQL schema visualization solutions. Both the versioned schemas defined and the diagrams proposed for their visualization are original and novel contributions of this paper. Moreover, two utilities have been implemented to support the diagrams presented here, which allowed us to experiment with the NoSQL schema visualization.

The remainder of this article is organized as follows. First, we briefly introduce the concept of aggregate-oriented data models and a running example. Then, we define the set of schemas identified and the diagrams proposed. Next, we comment the related work. Finally, we present the conclusions drawn from our work, which will guide our future work.

2 Aggregate-Oriented NoSQL Data Models

Object references and aggregate objects are constructs specifically conceived to represent complex data, but they are not part of the relational model. Aggregate objects are usually preferred to object references in the case of NoSQL databases, because the data is distributed through clusters to achieve scalability, and object references may involve contacting remote nodes. Thus, aggregate-orientation has been identified as a characteristic shared by the data models of the three most widely used NoSQL systems: key-value, document, and wide-column [14].

An aggregate-oriented NoSQL database can be seen as storing a set of semi-structured objects. Semi-structured data is mainly characterized by its schemaless nature. A semi-structured object O is composed of one or more fields (*attributes* or *properties*) p_i: $O = \{p_0, p_1, \ldots, p_m\}$. Each field p_i is specified by a pair $< n_i, v_i >$, where n_i and v_i denote the name and value of the field, respectively. The value of a field can be: (i) an atomic value (a number, string, boolean, ...); (ii) another object, i.e. an embedded object inside the object which the field belongs to; (iii) a reference to another object: This is usually a string or integer that matches the value of a field in the referenced object; or (iv) an array of values, which can be homogeneous or heterogeneous. Therefore semi-structured data has a hierarchical (nested) structure with a root object which can recursively embed other objects and arrays.

A database system stores data that relate to entities of real the world (i.e any physical or conceptual thing that exists). Here, an *entity* labels all the objects that refer to the same concept (e.g. movie, director, or prize). As indicated above, the schemaless nature of NoSQL databases allows for different stored objects of

the same entity type to have variations in their schemas. Therefore, we will introduce the notion of *entity version* to denote each of the sets of objects that, sharing the same entity label, have a different schema. Each entity will have one or more entity versions. Versions exist both for root and nested entities.

JSON [8] is a standard human-readable text format widely used to represent semi-structured data, and used in the majority of aggregate-oriented systems. Here we introduce a set of JSON objects that represent data about movies, which will be used throughout this article as an running example of database.

Our database example in Figure 1 is represented as an array that stores the collections of the *Movie* and *Director* entities. There are 3 versions of *Movie*, 2 versions of *Director* and *Criticism*, and one version of *Price* and *Rating*. *Movie* and *Director* are root entities; *Criticism*, *Rating*, and *Prize* are embedded into *Movie*. We refer to versions of an entity by means of the name of the entity joined to an unique version number by an underscore (e.g. *Movie_3* and *Director_1*)

We have supposed that a *Movie* entity has four common fields: *title*, *year*, *director*, and *genre*, and four optional fields: *prizes*, *rating*, *criticisms*, and *running_time*. The *Director* entity has two common fields: *name* and *directed_movies*, and the *acted_movies* optional field. The *Criticism* embedded entity has three common fields: *color*, *journalist*, and *media*, and one optional field, *url*. The *Prize* embedded entity has three fields: *year*, *event*, and *names*. Finally the *Rating* embedded entity has just two fields: *score* and *voters*.

Figure 2 shows the metamodel described in [15] to represent the information in an inferred schema of an aggregate-oriented data model. It would be formed by a set of entities that have one or more entity version schemas. Each entity version schema is composed by one or more properties or fields that are specified by its name and data type. These properties can be aggregation or reference relationships (i.e. associations), or attributes whose type can be a primitive type or tuple (i.e. an homogeneous or heterogeneous array of primitive types). Entities (and therefore entity versions) can be root or nested.

3 NoSQL Database Schemas

In this section, we shall define the types of schemas that we have identified for aggregate-oriented NoSQL databases. For each object in the database, an *Object Schema* (or *type*) is obtained by replacing, recursively, the atomic values of a semi-structured object by an identifier that denote its type (i.e. String, Number). Therefore, it has the same structure as the described object with respect to fields, nested objects, and arrays. The schema inference process starts with this set of *object schemas* to infer the set of entities and relationships.

In relational databases, an entity has just one schema, and the database schema is formed by the entities (its only schema) and the relationships between them. However, in NoSQL databases, the existence of *entity versions*, each with variations in their schema, adds a set of new dimensions to the schema definition.

An *entity version schema* (or simply *version schema*) is obtained from the object schema of an entity version by replacing each embedded and referenced

```
{ "rows": [
  { "type": "movie",
    "title": "Citizen Kane",
    "year": 1941,
    "director_id": "123451",
    "genre": "Drama",
    "_id": "1",
    "prizes": [
      { "year": 1941,
        "event": "Oscar",
        "names": [
          "Best screenplay",
          "Best Writing"
        ]},
      { "year": 1941,
        "event": "National Board of
            Review, USA",
        "names": [
          "Best Film",
          "Top Ten Films"
        ]}
    ],
    "criticisms": [
      { "journalist": "Roger Ebert",
        "media": "Chicago Sun-Times",
        "url": "http://chicago.
            suntimes.com/",
        "color": "green"
      },
      { "journalist": "Richard Brody"
          ,
        "media": "The New Yorker",
        "color": "green"
      }
    ]},
  { "type": "movie",
    "title": "The Man Who Would Be
        King",
    "year": 1975,
    "director_id": "928672",
    "genre": "Adventures",
    "_id": "2",
    "running_time": 129
  },
  { "_id": "3",
    "type": "movie",
    "title": "Truth",
    "year": 2014,
    "director_id": "345679",
    "genre": "Drama",
    "rating":{
        "score": 6.8,
        "voters": 12682
    },
    "criticisms": [
      { "journalist": "Jordi Costa",
        "media": "El pais",
        "color": "red"
      },
      { "journalist": "Lou Lumenick",
        "media": "New York Post",
        "color": "green"
      }
    ]},
  { "name": "Orson Welles",
    "directed_movies": ["1"],
    "acted_movies": ["1"],
    "type": "director",
    "_id": "123451"
  },
  { "type": "director",
    "directed_movies": ["3"],
    "name": "James Vanderbilt",
    "_id": "345679"
  },
  { "type": "director",
    "directed_movies": ["2"],
    "name": "John Huston",
    "_id": "928672"
  }
]}
```


Fig. 1: Movie Database for the Running Example.

objects by the corresponding name of the embedded or target entity version, respectively. These schemas can specify both root (*root version schema*) and embedded objects (*embedded version schema*). In Figure 3 we show (in JSON format) the root version schema for the *Movie_1* entity version.

A version schema therefore involves four kinds of entity properties: (i) *primitive* (*title*, *genre*, and *year*); (ii) *reference* (*director_id*); (iii) *aggregation*, (*Prize_1*); and (iv) *array type*, that can have one (homogeneous) or more (heterogeneous) base types that can in turn be primitive, aggregation, reference, or another array. The *criticisms* field is an array of either *Criticism_1* or *Criticism_2*. The *relationship type* term refers both to the reference and aggregation types. Note that these two types cause that a schema is either directly or indirectly related to others. The *Movie_1* version schema involves the *Criticism_1*, *Criticism_2*, *Director_1*, and *Prize_1* version schemas, and these schemas might refer in turn to other schemas, so that a schema graph is formed.

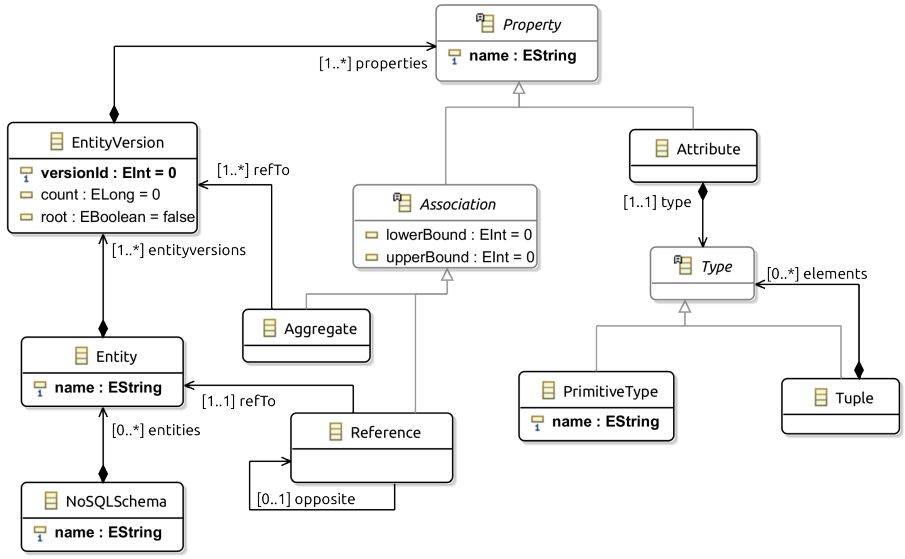

Fig. 2: *NoSQL-Schema* Metamodel Representing NoSQL Schemas.

```
{
    "title": "String",
    "year": "Number",
    "genre": "String",
    "director_id": "ref(Director)",
    "prizes": "Prize_1",
    "criticisms": [
        "Criticism_1",
        "Criticism_2"
    ]
}
```

Fig. 3: Schema for *Movie_1*.

```
{
    "title": "String",
    "year": "Number",
    "genre": "String",
    "director_id": "ref(Director)",
    "ratings": "Rating_1",
    "running_time": "Number",
    "criticisms": [
        "Criticism_1",
        "Criticism_2"
    ],
    "prizes": "Prize_1"
}
```

Fig. 4: Entity Union Schema for *Movie*.

An *entity schema* can be defined as the set of version schemas of a given entity. Transformations can be performed to this set for it to be useful for different purposes. For instance, sometimes it is interesting to have a "view" of all the version schemas of an entity. This can be done joining all the properties contained in the schemas of the entity versions of the entity: An *entity union schema* could be constructed with the following rules:

1. For each property whose name appears only in one entity version schema, add that property to the *entity union schema*.
2. For each property whose name appears in more than one entity version schema:

(a) If the type of the property is the same in all the entity version schemas in which it appears, add that property to the *entity union schema*.
(b) If the type of the property differs in some entity version schemas, collect the set of different types of the property and build a *union type*. A union type of two types T_1 and T_2, denoted as $U(T_1, T_2)$, can be defined as a type that describes both the elements described by T_1 *and* those described by T_2.

Figure 4 shows the entity union schema for the *Movie* entity of our database in JSON format.

The usual schema inference for JSON objects in NoSQL systems discovers schemas that result of the union of the object schemas of each entity version, but version schemas and entity schemas are not discovered. Therefore the structure of the schema is similar to those defined for object schemas. We refer to these schemas as *union object schemas*. Several strategies are used to solve the problem of conflicting types when a field belongs to more than one version. For instance, the type of the field can be the union of all the types encountered, can be promoted to the *Object* type [9] or to the String type [13,7].

Finally we shall define two kinds of schemas for aggregate-oriented databases that involve all the entities:

- *Database schema*. It is formed by the set of *root version schemas* that describe the root entity versions of the database. As a schema recursively depends on the schemas of the embedded or referenced entities, a complete schema is formed by the set of version schemas of all the entity versions that exist in the database.
- *Entity database schema*. It is formed by the entity union schemas that describe all the root entities of the database.

Next we show how the defined types of schemas could be visually represented. We will first consider UML class diagrams. These diagrams together with ER diagrams are traditionally used to represent conceptual and logical relational schemas. Here, we will show their limitations for NoSQL schemas and the need for the definition of a specific notation.

4 NoSQL Schema Visualization with UML Class Diagrams

PlantUML [12] is a drawing tool for visualizing UML diagrams. It provides a textual language to express the diagrams, that is transformed into DOT code to be rendered by Graphviz [6]. Using the PlantUML notation we can define formatting features of diagrams, such as colors and icons for elements. We have used PlantUML to visualize the three kinds of schemas that can be represented by using UML class diagrams: version schemas, union entity schemas, and entity database schemas. Each kind of diagram has been automatically generated by means of a model-to-text transformation that generates the PlantUML code that corresponds to the input NoSQL-Schema model.

Version Schemas. A version schema can be shown as a UML class diagram. For instance, Figure 5 shows the root version schema for the *Movie_3* entity version. Each entity version is represented as a class, and a letter within a small circle is used to distinguish the root entity ("R") from the embedded entity versions ("V"). The class name is the entity version name in the inferred schema model. All the version schemas directly or indirectly nested to the root entity version are shown by means of unidirectional composite relationships whose name and cardinality are the same than the corresponding aggregation elements of the schema model.

As explained in [15], the target of a reference is an entity, not an entity version, therefore a version schema can also include entities, and the "E" letter is used to label the classes that represent entities. For entities, the diagram shows the embedded or referenced entities but not entity versions. If an entity references the root entity version then the reference links are not shown, but the specification is enclosed after the attribute list in the format: the keyword *ref* followed by the entity name and the property name.

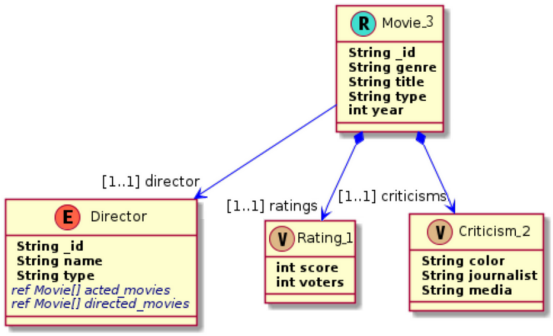

Fig. 5: Root Entity Version Schema for *Movie_3* Generated with PlantUML.

With this representation, heterogeneous arrays cannot be displayed directly. We could generate, for example, artificial "union" classes to join all the included classes in heterogeneous arrays, similarly as how we constructed the entity union schemas, but this would be more visually intrusive. Or we could generate N aggregation relationships with numbered suffixes to each of the included classes. In any case, the hierarchical structure of these schemas can be appropriately represented with class diagrams.

Entity Union Schemas. We have followed the strategy explained above for visualizing entities in root version schemas. When several version schemas have a property with identical name but different type, the union type inferred for this property can only be visualized if it has only two versions: one includes the property with a primitive type (or a tuple) and the other one is a relationship, but the rest of possible unions of types would cause an error because would have several attributes (or relationships) with the same name and different type (or

association end). Figure 6 shows the union schema for the *Movie* entity, which includes six attributes, three composite relationships for the *Criticism*, *Prize*, and *Rating* union schemas, and a reference relationship to the *Director* entity union schema. The diagram also shows the relationships of the aggregated or referenced union schemas, for instance, two reference relationships from *Director* to *Movie*. While the previous diagram showed all the referenced or embedded entity versions involved in the definition of a version schema for a root entity, the diagram for an entity union schema only includes entities. Therefore, in the diagram in Figure 6 the reference relationships from *Director* to *Movie* are shown because no confusion is caused.

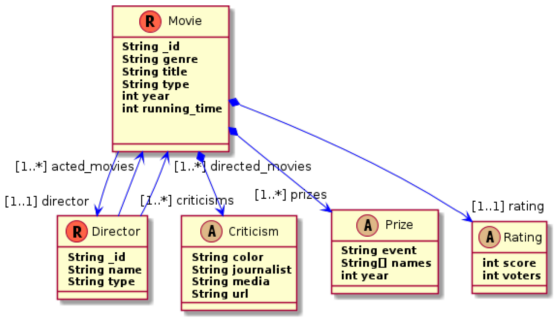

Fig. 6: Entity Union Schema for Movie Generated with PlantUML.

The main limitation of class diagrams for version schemas is the representation of union types, as several attributes with the same name but different type are not allowed. Note that the graph-like structure of a entity union schema can be adequately represented by class diagrams.

Entity Database Schemas. A diagram of the entity database schema is formed by superposing all the diagrams for root union schemas. It is worth to note that an entity union schema can be directly or indirectly connected to the rest of entity union schemas existing in the database, and then such a schema will be equivalent to the entity database schema. In our running example, this diagram would be the same as that shown in Figure 6.

5 Visualization of NoSQL Schemas with Specific Notation

Tackling the visualization of NoSQL schemas with UML class diagrams evidenced the convenience of defining a specific notation for this purpose because UML class diagrams cannot represent entity schemas or database schemas, and visualizing the rest of schemas has the limitations commented in the previous section. Moreover, the cases in which "sets" of schemas are generated, some kind of browsing capability is needed to navigate the structure of the database.

Therefore, we have developed a diagramming tool that has been specially designed to visualize NoSQL schemas. This solution took advantage of the fact that our inference process generates models that conform to an Ecore metamodel to significantly reduce the development time and effort in relation to create the editor from scratch. We have used Sirius [16], a robust and powerful tool aimed to define graphical notations for existing metamodels. The process is as follows: First, we define the notation for the NoSQL-Schema metamodel by using the capabilities offered by Sirius for this task. Taking as input the metamodel and its notation, Sirius generates (i) an editor to create and visualize diagrams of models that conform to the input metamodel, and (ii) a model injector that generates a model from the graphical representation. Next, we will comment the main features of the different diagrams and views created. The tool is available on the Sirius Gallery.[1]

Global View Tree. This view shows a tree with three branches: *Schemas*, *Inverted Index*, and *Entities*, as illustrated in Figure 7. *Schemas* list all the root entities with their version schema; given a root version schema, the user can browse their embedded and referenced schemas. For instance, Figure 7 shows the aggregated and referenced schemas from the root version schemas of *Movie* and *Director*. Schemas for *Director_1* are shown. *Inverted Index* lists an inverse index of versions. This kind of index has been defined to navigate from a root or embedded version schema to all the root version schemas from which it is referenced (for example, *Director_1* is referenced from *Movie_1*.) *Entities* list all the entities that exist in the database. Both root and embedded entities are included in the list. The user can select an entity to display its entity versions, and then he or she can inspect their properties and types. These three branches show, in different form, the information included in the *database schema* as defined in Section 3.

In this tree, entities, entity versions, and root entity versions are represented with indicative icons. Also attributes, aggregation and reference relationships. These icons are used in all the diagrams created to provide a uniform view to the user. It is possible to navigate from the Global View Tree to the other diagrams by means of contextual menus.

Database Schema Diagrams represent the information included in a database schema as shown in Figure 8. With this diagram, the user can see at a glance (i) the database entities, (ii) the set of version schemas of each entity, and (iii) the attributes and relationships of each version schema. Version schemas are visualized as rectangles nested into a rectangle that represents the root or embedded entity schema it belongs to. Aggregation and reference relationships are visualized as a solid line and are tagged as *aggregates* and *references* respectively.

Entity Schema Diagrams represent an entity schema as shown in Figure 9. They are visualized just like database schemas, but with only one entity schema shown, which can correspond to a root or embedded entity. The Figure shows that

[1] `https://eclipse.org/sirius/gallery.html`.

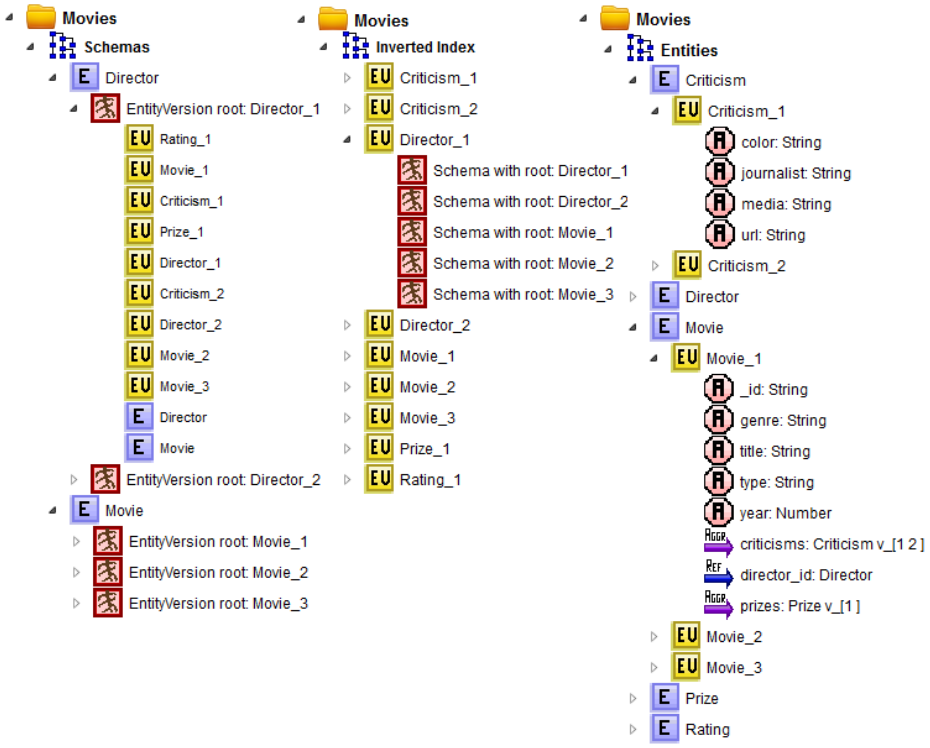

Fig. 7: Global Schema Tree for the *Movie* example.

the *Movie* entity has three versions, and the corresponding version schemas are shown. For instance *Movie_ 1* aggregates a version schema of the *Prize* entity, as well as two version schemas of the *Criticism* entity, among other references.

Root version schemas and Entity union schemas are visualized with diagrams similar to those generated with PlantUML in Figures 5 and 6.

6 Related Work

Some well-known modeling tools for relational databases have been recently augmented to offer functionality for NoSQL databases. ER/Studio Data Architect [4] and DBSchema [3] support a schema discovery process for MongoDB [10], and visualization of inferred schemas as E/R-like diagrams. The schema discovery process is applied on a single collection, and the diagram obtained shows the union entity schema, but references are ignored, that is, the diagram visualizes the root entity and the directly or indirectly embedded entities. DBSchema allows the references to be added once the discovery task is performed. At this moment, these tools do not support functionality related to the visualization or browsing of versioned schemas, with the exception of the entity union schemas.

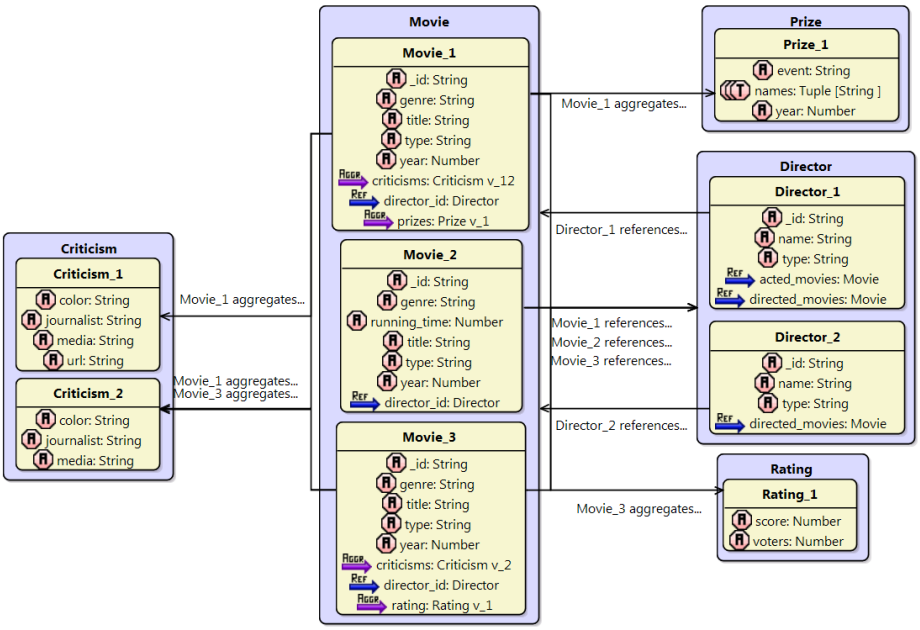

Fig. 8: Database Schema Diagram for the *Movies* Database.

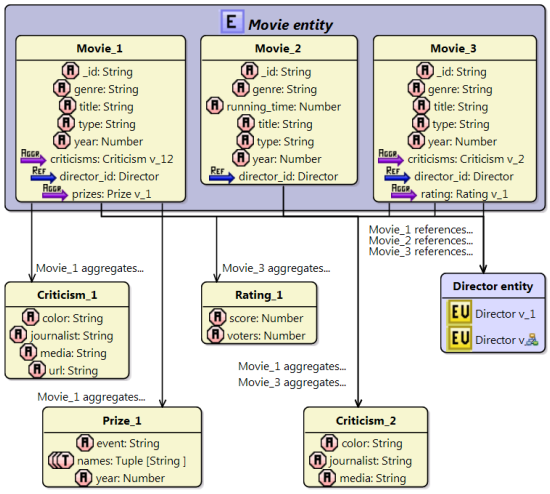

Fig. 9: Entity Schema Diagram for *Movie*.

ERwin Unified Data Modeler is a project under development with the aim of supporting data modeling for relational and NoSQL systems [5]. This tool will provide data schema discovery and data migration between RDBMS and NoSQL databases. A data model based on the E/R notation is used to represent

relational and NoSQL logical schemas in a unified way. Physical models have been also defined, which can be automatically generated from a logical model. A logical model can be automatically extracted and visualized from data. Query and data production patterns are used to transform logical models into physical models. However, details about the implementation of this tool are not provided and versioned schema visualization is not considered.

ExSchema [2] aims to discover schemas for NoSQL stores by applying a static analysis of the source code of applications that use data management APIs. Discovered schemas are visualized as PDF images that have a large amount of visual elements. Aggregation and reference relationships are not explicitly differentiated, which makes it difficult to visually identify entities and relationships. Neither versioned schemas nor entity union schemas are addressed.

JSON Discoverer is a tool aimed to infer JSON schemas from JSON-based Web APIs [7]. Domain models generated correspond to the notion of entity database schema defined in Section 3, but excluding references between entities. Schemas are drawn as UML class diagrams. Unlike the tools previously commented, JSON discoverer extracts entity domain models rather union object schemas, which requires discovering and extracting the involved aggregation relationships. However, the existence of data versions (i.e. versioned schemas) is not addressed, and the references between objects are not discovered. Note that this tool is not focused on database schema visualization.

7 Conclusions and future work

Modeling tools for aggregate-oriented NoSQL systems should be designed taking into account the existence of entity versions. In this article we have explored the schemas that could be useful to designers and developers and how they could be visually represented. We have observed that the diagrams proposed for version schemas and entity union schemas are appropriate and the limitations exposed for UML class diagrams are solved in the Sirius solution. For entity schemas, in addition to entity union schemas and browsing version schemas, a tree representation that explicitly shows the common and specific properties among the version schemas could be very useful, and will be studied as a future work. The database schema diagram shown in Figure 8 is difficult to understand due to the existence of relationships between entity versions, and may become infeasible when entities have more than a few entity versions. Therefore, database schemas require browsing capabilities and even query languages if the number of schema versions is high. This is also planned for future work. For entity database schemas, UML class diagrams are adequate, but properties with the same name but different type should be allowed. The order of magnitude of the number of version schemas for entities significantly affects to the mechanisms implemented around the diagram representation. We have tested some open datasets such as Stackoverflow, and have found that around a hundred schema versions is common, but there are also cases with a very large number of schema versions (tens of thousands) [17]. Therefore, a research effort should be devoted to study

the properties needed for the representations to be effective, including, but not being limited to: i) having views to show version specific properties as well as entity common properties, ii) allowing the user to establish optional fields (diminishing the number of versions), and iii) defining methods to order the views using statistics such as the number of objects per version, in order to detect erroneous or undesired versions. Regardless, browsing capabilities are needed to navigate through entity and version schemas. The visualization tools presented are available in a GitHub repository.[2]

References

1. Bacvanski, V., Roe, C.: Insights into NoSQL Modeling: A Dataversity Report. DataVersity (2015)
2. Castrejón, J.C., Vargas-Solar, G., Collet, C., Lozano, R.: ExSchema: Discovering and Maintaining Schemas from Polyglot Persistence Applications. In: 2013 IEEE ICSM. pp. 496–499 (2013)
3. DbSchema: Dbschema web page. `http://www.dbschema.com/index_es.html` (Visited March 2017)
4. ER-Studio Web Page. `https://www.idera.com/er-studio-enterprise-data-modeling-and-architecture-tools` (2017), accessed: March 2017
5. CA ERwin Web Page. `http://erwin.com/products/data-modeler` (2017), accessed: March 2017
6. GraphViz: GraphViz Visualization Software. `http://www.graphviz.org/` (Visited: March 2017)
7. Izquierdo, J.L.C., Cabot, J.: JSONDiscoverer: Visualizing the Schema Lurking Behind JSON Documents. Knowl.-Based Syst. 103, 52–55 (2016)
8. JSON: JavaScript JSON. `http://www.w3schools.com/js/js_json.asp` (Visited March 2017)
9. Klettke, M., Scherzinger, S., Störl, U.: Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In: BTW. vol. 2105, pp. 425–444 (2015)
10. MongoDB Web Page. `https://www.mongodb.com/` (2017), accessed: March 2017
11. NoSQL Market. `https://www.alliedmarketresearch.com/NoSQL-market` (Visited November 2016)
12. PlantUML: PlantUML in a nutshell. `http://plantuml.com/` (Visited: March 2017)
13. Rückstieß, T.: `mongodb-schema` npm package. `https://www.npmjs.com/package/mongodb-schema` (2016), visited April 2016
14. Sadalage, P., Fowler, M.: NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley (2012)
15. Sevilla Ruiz, D., Feliciano Morales, S., García Molina, J.: Inferring Versioned Schemas from NoSQL Databases and Its Applications. In: 34th International Conference, ER2015, Stockholm, Sweden. pp. 467–480 (2015)
16. Sirius: Sirius official website. `https://eclipse.org/sirius/` (Visited March 2017)
17. Wang, L., Hassanzadeh, O., Zhang, S., Shi, J., Jiao, L., Zou, J., Wang, C.: Schema Management for Document Stores. In: VLDB Endowment. vol. 8 (2015)

[2] `https://github.com/catedrasaes-umu/NoSQLDataEngineering`.