Inferring Versioned Schemas from NoSQL Databases and Its Applications

Diego Sevilla Ruiz(✉), Severino Feliciano Morales, and Jesús García Molina

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

Abstract. While the concept of database schema plays a central role in relational database systems, most NoSQL systems are schemaless: these databases are created without having to formally define its schema. Instead, it is implicit in the stored data. This lack of schema definition offers a greater flexibility; more specifically, the schemaless databases ease both the recording of non-uniform data and data evolution. However, this comes at the cost of losing some of the benefits provided by schemas. In this article, a MDE-based reverse engineering approach for inferring the schema of aggregate-oriented NoSQL databases is presented. We show how the obtained schemas can be used to build database utilities that tackle some of the problems encountered using implicit schemas: a schema diagram viewer and a data validator generator are presented.

Keywords: NoSQL databases · Schemaless databases · Schema inference · Model-driven data reverse engineering · JSON

1 Introduction

Modern applications that have to deal with huge collections of data have evidenced the limitations of relational database management systems. This has motivated the development of a continuously growing number of non-relational systems, with the purpose of tackling the requirements of such applications. Specially, the ability to represent complex data and achieving scalability to manage both large data sets and the increase in data traffic. The NoSQL (*Not SQL/Not only SQL*) term is used to denote this new generation of database systems.

The lack of an explicit data schema (*schemaless*) is probably the most attractive NoSQL feature for database developers. While relational systems require the definition of the database schema in order to determine the data organization, in NoSQL databases data is stored without the need of having previously defined a

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/).

P. Johannesson et al. (Eds.): ER 2015, LNCS 9381, pp. 467–480, 2015.
DOI: 10.1007/978-3-319-25264-3_35

schema. Being schemaless, a larger flexibility is provided: the database can store data with different structure 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, since a schema is implicit into data and database applications. The developers must always keep in mind the schema when they write code that accesses the database. For instance, they have to honor the names and types of the fields when writing insert or query operations. This is an error-prone task, more so when the existence of several versions of each entity is probable. Therefore, the idea is emerging of combining a schemaless approach with mechanisms (e.g. data validations against schemas) that guarantee a correct access to data [6,10]. On the other hand, some NoSQL database tools and utilities need to know the schema to offer functionality such as performing SQL-like queries or automatically migrating data. A growing interest in managing explicit NoSQL schemas is therefore arising [9,10,12,15].

This article presents a reverse engineering strategy to infer the implicit schema in NoSQL databases, which takes into account the different versions of the entities. We call these schemas *Versioned Schemas*. The usefulness of the inferred versioned schemas is illustrated through two possible applications: schema visualization, and automated generation of data validators. The approach has been designed to be applied to NoSQL systems whose data model is aggregate-oriented [13], which is the data model of the three most widely used types of NoSQL stores: *document*, *key-value*, and *column family* stores. Model-Driven Engineering (MDE) techniques, such as metamodeling and model transformations, have been used to implement both the schema inference strategy and the applications, in order to take advantage of the abstraction and automation capabilities that they provide.

There are therefore two main contributions in this work. To our knowledge, this is the first approach that infers conceptual schemas from NoSQL databases discovering all the versions of the inferred entities and relationships. Moreover, we show how the inferred schemas can be used to automatically generate different software artifacts, which help to improve the productivity and code quality. The approach proposed has been validated for the MongoDB, CouchDB, and HBase stores, and the tools implemented may be downloaded from http://www.catedrasaes.org/wiki/NoSQLSchemaVersions.

This article is organized as follows. The following Section explains the notion of aggregate-oriented data model, and presents a running example. Section 3 gives an overview of the approach proposed, and Sect. 4 describes in detail the schema inference strategy. The utilities that have been built are described in Sect. 5, and the related work is discussed in Sect. 6. Finally, conclusions and future work are presented in Sect. 7.

2 Background

This section introduces some key concepts that are used throughout the article and motivates the work. Moreover, a simple NoSQL database is shown, which will be used as a running example.

2.1 Semi-structured Data and the JSON Format

Semi-structured data is mainly characterized by the fact that it has a non-uniform and an implicit structure, which can evolve rapidly [1]. This data is expressed in formats, such as XML and JSON, which allow the representation of information in hierarchical form (i.e. a tree-like structure) by using tags or symbols as separator elements.

JSON (JavaScript Object Notation)[1] is a standard human-readable text format widely used to represent semi-structured data. This notation is taking the place of XML as primary data interchange format because it is more simple and legible. A JSON object or document is formed by a set of key-value pairs (*fields*). The type of a JSON value may be a primitive type (Number, String, or Boolean), an object, or an array of values. `null` is used to indicate that a key has no value.

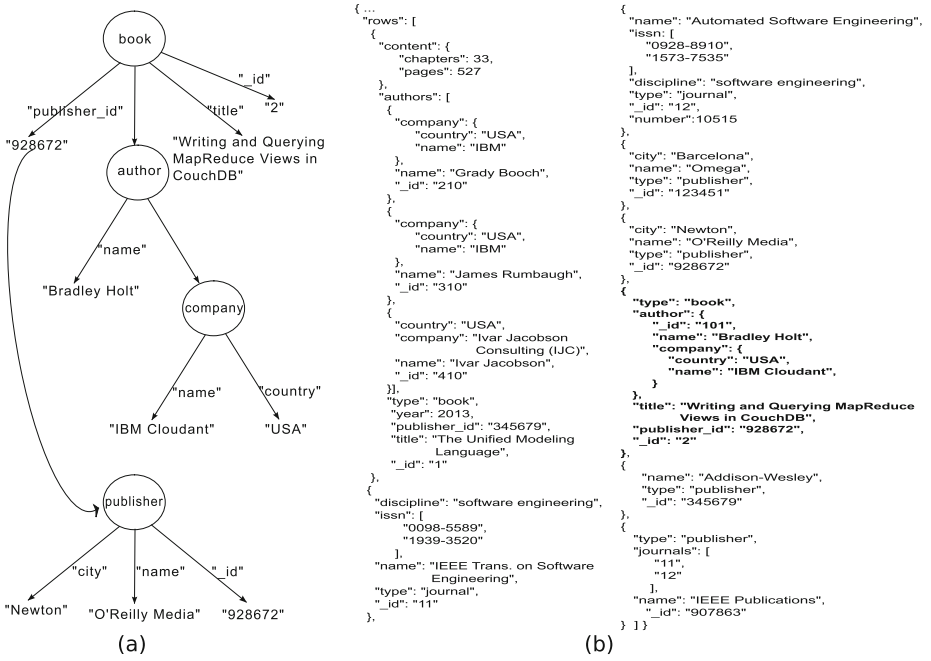


Fig. 1. Example database used and a tree representation of a book.

As indicated in [4], a piece of semi-structured data can be formalized as a tree whose leaf nodes are atomic values of primitive types (e.g. string, integer, float, or boolean) and the root and intermediate nodes are objects (i.e. tuples) or either arrays of objects or values. The edges are labelled with the names of the attributes. A root or intermediate node has a child node by each attribute of the object associated. For instance, Fig. 1(a) shows the tree that corresponds to the JSON object that represent the book with *_id=2* in Fig. 1(b). References

[1] http://json.org/.

among data may be expressed in a similar way as foreign keys in relational databases, that is, the atomic value of an attribute (e.g. *publisher_id* in *book*) matches a value in another attribute of a different object (e.g. *_id* in *publisher*).

The term *aggregate* is normally used to refer to the object structure that consists of a root object that recursively embeds other objects, so that the tree-like structure of an semi-structured data is aggregate-oriented. In Fig. 1(a), the *book* object aggregates an *author* object, which aggregates, in turn, a *company*.

2.2 Aggregate-Oriented Data Models

While complex data is addressed in relational databases through joins by means of foreign keys (i.e. references between tables), object references and aggregate objects are more appropriate ways to represent such data. Unlike object-oriented databases, 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 [13]. They organize the storage in form of collections of key-value pairs in which the values can also be collections of key-value pairs, and the "aggregate-oriented data model" has been proposed to refer these three data models.

Actually, these systems store semi-structured data, and they are therefore characterized by the fact that explicit schemas do not have to be defined to specify the structure of the data, which provides them greater flexibility. Thus, data that has different structure for the same entity could be stored (i.e. non-uniform data). The evolution of the data is also easier because there is no need of a schema evolution, and different versions of data can coexist.

The absence of a schema, however, has some drawbacks for developers and tool implementors. When a schema is formally defined (e.g. a relational schema), a static checking assures that only data that fits the schema can be manipulated in application code, and mistakes made by developers in writing code are statically spotted. In fact, the analogy to statically and dynamically typed languages is commonly used to note the difference among databases with and without a schema [4]. On the other hand, a number of tools need the information contained into schemas in order to implement their functionality (e.g. query engines and validator generators). Therefore, an increasing attention is being paid to the topic of the NoSQL schema inference and some approaches has been proposed [12,15].

A schema of an aggregate-oriented data model is basically formed by a set of entities connected through two types of relationships: aggregation and reference. Each entity will have one or more fields that are specified by its name and its data type. Several versions of an entity can exist due to the non-uniformity characteristic and the database evolution.

2.3 A NoSQL Database for the Running Example

For the purposes of this work, we consider a NoSQL database as an arbitrarily large array (i.e. a collection) of JSON objects that include: (a) a field (e.g. *type*)

that describes its entity type; and (b) some form of unique identifier for the object (in our case the *_id* field). This format is non-compromising, and provides system independence. In fact, it is very similar to what it is actually used in most NoSQL database implementations. For example, CouchDB guides recommend the usage of the *type* field. MongoDB creates one collection for each type of object, so that the collection name could provide the *type* field. In HBase, the *type* field of an object could be the name of its *column family*. If the value of the *type* field is not directly obtainable, some heuristics could be used. However, in some cases it may require the user to provide it.

Figure 1(b) shows a simple database that stores objects for the *Book*, *Publisher*, and *Journal* entities. The first *Book* object aggregates an array of authors (*authors* field) and an embedded object for the content (*content* field). In turn, the *author* field aggregates an embedded object that records the company for which he or she works (*company* field). With regards to object references, both *Book* and *Publisher* objects show examples of them. The *Book* objects have a reference to its publisher (*publisher_id* field) and *Publisher* objects hold a reference to the list of journals published (*journal* field).

It is worth noting that aggregates and references are implicit: a parser could identify the embedded objects. However, references require some kind of heuristics if conventions are not used. Some idioms have been therefore proposed to express references in NoSQL databases, some of them considered in Sect. 4.

3 Overview

We shall here outline the general arquitecture of the proposed approach to infer schemas. Moreover, we shall describe the metamodel created to represent NoSQL schemas.

Reverse engineering can take advantage of MDE techniques. Metamodels provide a formalism to represent the knowledge harvested at a high-level of abstraction, and automation is facilitated by using model transformations. Therefore, we have devised an MDE solution to reverse engineer versioned schemas from aggregate-oriented NoSQL databases that we use to create database utilities.

Figure 2 shows the architecture of the solution, which is organized in three stages. Firstly, a Map-Reduce operation is applied in order to extract a collection of JSON objects that contains one object for each version of an entity, i.e. the minimum number of objects that are needed to perform the inference process. Map-Reduce is germane to most NoSQL databases, and gives an advantage in performance, as it is the native processing method when an algorithm has to deal with all the objects in a database. Secondly, that collection is injected into a model that conforms to a JSON metamodel, which is easily obtained by mapping the JSON grammar elements into the metamodel elements. Thirdly, the reverse engineering process is implemented as a model-to-model transformation whose input is the JSON model, and that generates a model that conforms to the NoSQL-Schema metamodel (Fig. 3). The inferred NoSQL-Schema models may be used to build tools that could be classified in two categories: (i) database utilities

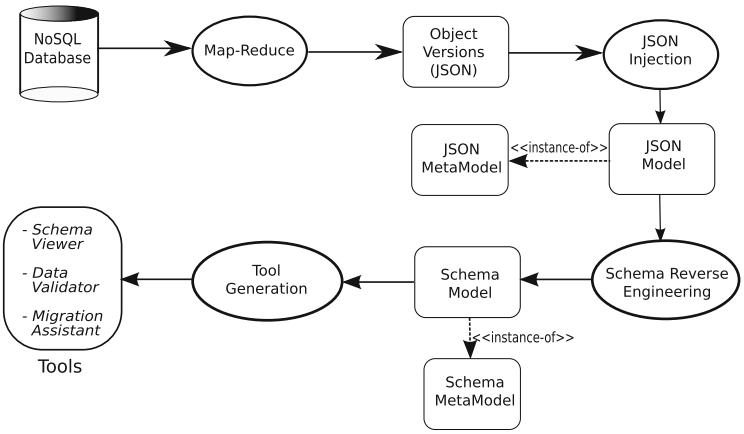


Fig. 2. Overview of the proposed MDE architecture.

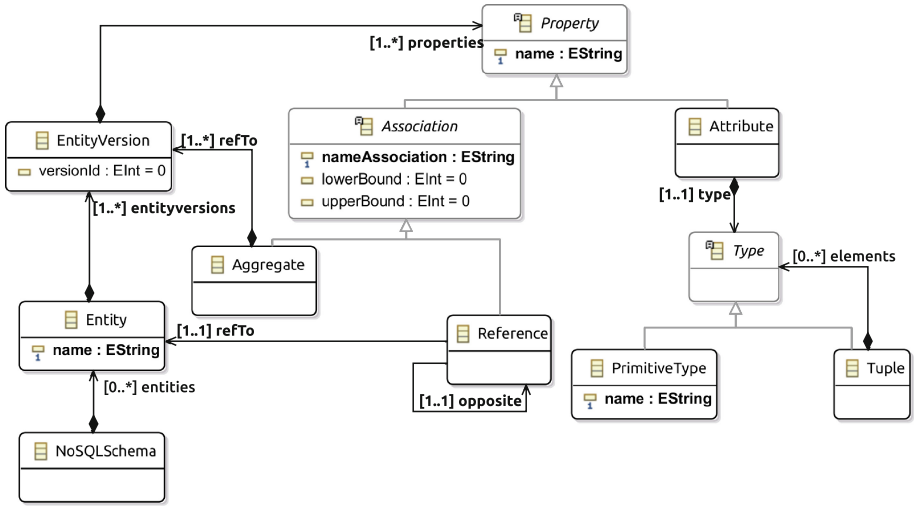


Fig. 3. *NoSQL-Schema* metamodel representing NoSQL schemas.

that require knowledge of the database structure, and (ii) helping developers to deal with problems caused by the absence of an explicit schema, for instance the tools presented below, which are able of generating data validators, migration scripts or schema diagrams.

Figure 3 shows the NoSQL-Schema metamodel that represents schemas of aggregate-oriented NoSQL databases according to the notion of NoSQL schema presented in Sect. 2.2. A schema (metaclass *NoSQLSchema*) is formed by a collection of entities (*Entity*); one or more versions (*EntityVersion*) exist for each entity. A version is defined by a set of properties (*Property*) that may be *Attributes* or *Associations*, depending on whether the property represents a type (either a *PrimitiveType* or a *Tuple*) or a relationship between two entities.

A tuple denotes a collection that may contain primitive types and tuples. An association can be either an *Aggregation* or a *Reference*. The cardinality of an association is captured by the *lowerBound* and *upperBound* attributes, which can take values 0, 1, and −1.

Note that an aggregate is connected to one or more entity versions (*[1..*]refTo*) because an embedded object may aggregate an array with objects of different versions. Instead, a reference is connected to one entity (*[1..1]refTo*), since we need to know that a version holds references to a certain entity, but we decided not to cross object boundaries. The *opposite* self-reference in the *Reference* metaclass is used to make the relationship bidirectional, and specifies the other end.

4 Reverse Engineering Process

Extracting Versioned Schemas from aggregate-oriented NoSQL databases involves discovering entities, versions of each entity, the attributes of each version, and relationships between entities (aggregations and references). The reverse engineering algorithm should traverse all the stored objects (i.e. root entities), and analyze their properties in order to identify all the schema elements.

4.1 Building the *Raw Schema* of an Object

The first step in discovering the versioned schemas is obtaining what we call the *raw schema* of an object, which is a JSON object built honoring two rules: (i) it has the same structure as the described object with respect to fields, nested objects and arrays, and (ii) each primitive value in the described object is substituted in the raw schema by its JSON type (e.g. String or Number).

In our running example (Fig. 1(b)), {*name:String, city:String*} would be the raw schema for the *Publisher* entity with *_id=123451*, and {*title:String, publisher_id:String, author:*{*name:String, company:*{*country:String, name:String*}}} would be the raw schema of the *Book* with *_id=2*. More visually:

JSON object	**Raw Schema**
{name:"Omega", city:"Barcelona"}	*{name:String, city:String}*
{title:"Writing and...", *publisher_id:"928672",* *author:{name:"Bradley Holt",* *company:{country:"USA",* *name:"IBM Cloudant"} } }*	*{title:String,* *publisher_id:String,* *author:{name:String,* *company:{country:String,* *name:String} } }*

4.2 Obtaining the Version Collection

To improve the efficiency, we have considered a preliminary stage to the reverse engineering process. In this stage, a Map-Reduce operation is applied to obtain

a collection that only contains one object for each entity version, which will be referred to as the *Version Collection*.

For each object, the `map()` operation performs a two-step process. First, it generates the *version identifier*: the string obtained by concatenating the value of the special *type* field with a textual representation of the object's raw schema. Secondly, the <*version identifier, object*> key/value pair is emitted.

Then, the `reduce()` operation is performed once for each version identifier. It receives a set of objects that share the same version identifier and selects one of the objects as the archetype for the group, adding it to the output list. The result is an array of JSON objects following the format explained in Sect. 2.3, and shown in Fig. 1(b), but now containing just one object per object version.

4.3 Obtaining the Schema

The JSON object collection obtained in the previous stage is injected into a JSON model, from which a model-to-model transformation generates the Schema model. The transformation discovers the elements of the schema, and works as follows:

Discovering Entities and Entity Versions. For each JSON object in the model, an *EntityVersion* is considered. This usually leads to a new *EntityVersion*, but not in all cases, because a similar *EntityVersion* may exist already that only differs with the considered one with respect to cardinalities. If this is the case, the cardinalities of the existing *EntityVersion* are adjusted to include both specifications, and no new *EntityVersion* is created. When the created *EntityVersion* is the first one discovered for a particular entity, an *Entity* element is also generated. Each *Entity* holds a list of entity versions, in which each new *EntityVersion* is added. Obtaining an entity name differs for root and embedded objects. For *root objects*, the name is given by the *type* field of the object; for *embedded objects*, the name is given by the key of a pair whose value is a JSON object. If the value is an array of objects and the name is plural, then the singular name is used.

An *EntityVersion* is named by appending, to the entity name, a suffix with an underscore and a counter of the number of version. For instance, three *EntityVersion* would be generated for the *Publisher* root objects of the running example, named *Publisher_1*, *Publisher_2*, and *Publisher_3*, and *Author_1* and *Author_2* would be generated for the *Author* embedded object. Figure 4(b) shows a textual report with all the entity versions.

Discovering Attributes. An *Attribute* is generated for each object's pair whose value is either atomic or an array of either primitive types or nested arrays of primitive types. The attribute name is given by the pair name. With regard to the type, a *PrimitiveType* or a *Tuple* is generated depending on whether the value is atomic or an array. Each created *Attribute* is added to the collection of attributes of the corresponding *EntityVersion*. For instance, the pair *"title": "The Unified Modeling Language"* in a version of *Book* would lead to the *Attribute* named

"title" and a *PrimitiveType* named "String"; and the pair *"issn": ["0928-8910", "1573-7535"]* in a version of *Journal* would generate an *Attribute* named "issn" and a *Tuple*, as shown in Fig. 4(b).

Discovering Aggregation Relationships. A pair results in an *Aggregate* (i.e. an aggregation relationship) if its value is an object. Each created *Aggregate* must be connected to the *EntityVersion* that corresponds to the object value. Several *EntityVersion*s may embed the same aggregated *Entity*. For this, the transformation is organized in two stages. Firstly, *Entities*, *EntityVersion*s, *Attribute*s, *Type*s and *Pair*s are created, and then *Aggregate*s and *Reference*s are created in a second stage, once all the *EntityVersion*s have been created.

Regarding to the cardinality, the *lowerBound* and *upperBound* attributes take their values depending on the multiplicity of the *Pair*, e.g. it is *one-to-one* (*lowerBound=1* and *upperBound=1*) if the pair value is an object that can be *null*, and the cardinality is *zero-to-many* (*lowerBound=0* and *upperBound=−1*) if the pair value is an array of objects that can take the *null* value.

The *Aggregate* name is the same as the pair name but there are some exceptions. For instance, if the cardinality is *zero-to-many* or *one-to-many*, a singular name is converted into a plural name. In the database example, the *Book_1* entity version would aggregate several *Author*s, so the *Aggregate* name would be *authors*, with cardinality *one-to-many*. The three aggregation relationships discovered for the running example are shown in the diagram of the Fig. 4(a).

Discovering Reference Relationships. A reference implies that a entity's pair identifies to an object of another entity, that is, the pair values of the referencing entity match the values of another pair in the referenced entity. These identifier values are Strings, Integer numbers or arrays of these two primitive types. Two strategies are applied to discover references (i.e. *Reference* elements):

- Some conventions commonly used to express references are checked, such as:
  - If a pair name has the *entityName_id* suffix, then, a entity named *entityName* would be referenced if it exists.
  - MongoDB itself suggests to use a construct like {*$ref:"entityName", $id:"reference_id"*} to express references to objects of the entity named "entityName" [11].
- If a pair name is the name of an existing entity and the pair values match the values of a *_id* pair of such an entity.

For instance, the *journal* field of a *Publisher* version references to an array of *Journal* objects and the *publisher_id* field of a *Book* version references a *Publisher* object (Fig. 4(a)).

As in the case of aggregations, the references are connected to the corresponding entity in the second stage of the transformation. The cardinality is obtained for references as explained above for the aggregation relationships. Once all the references have been generated, the *opposite* relationship is resolved.

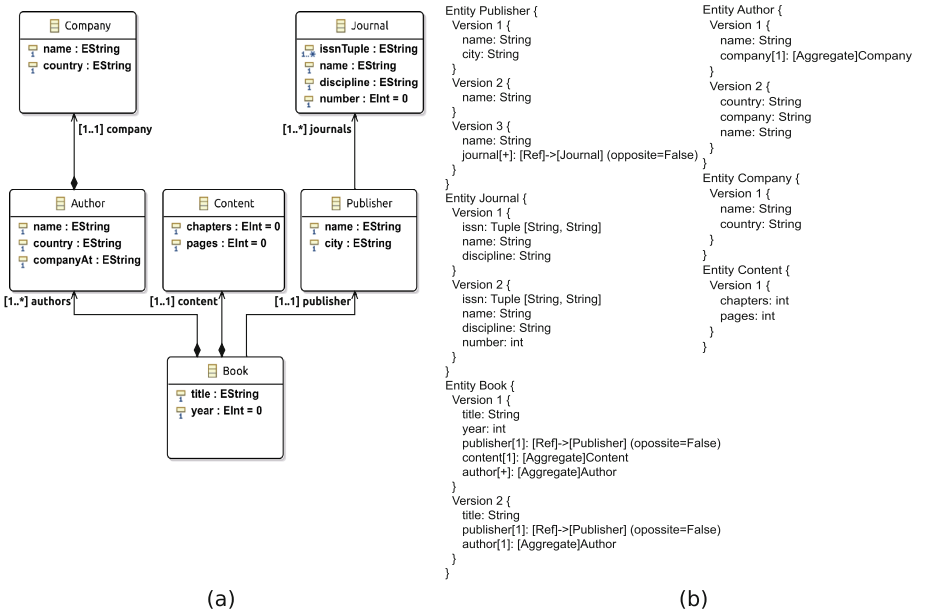
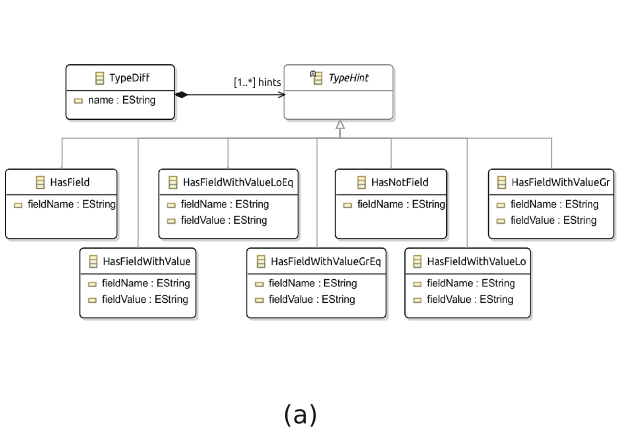


Fig. 4. Graphical representation of all the entities with the sum of all fields, and the textual report of versions.

5 Versioned NoSQL-Schema Applications

The inferred schemas are useful to build a number of tools intended to help developers that make use of NoSQL databases. There are tools that require knowledge of the schema in order to provide certain functionality (e.g. SQL query engines). On the other hand, the schema inference may be used to mitigate the problems due to the lack of an explicit schema. For instance, reports, diagrams, validators, and version migration scripts could be automatically generated from the NoSQL Schema models. As a proof of concept, we have created a schema viewer and a validator generator in order to illustrate the possible applications of the inferred schemas. We shall describe these utilities in this Section, and other applications will be outlined in Sect. 7.

Several benefits are gained by representing NoSQL schemas: both reasoning about them and its communication are facilitated, and a documentation separated from the code is obtained. Figure 4(b) shows a textual report of all the entity versions in our running example; attributes (name and type), and aggregate and reference relationships are indicated for each entity version. These reports are automatically generated by a model-to-text transformation that has the schema model as its input. As shown in Fig. 4, the inferred schemas have been also visualized as UML class diagrams. Entities are shown as classes, field as attributes, aggregate as composite associations, and references as associations that are navigable at the end owned by the referenced entity. Note that entity




```
function isOfExactTypeBook_2(obj) {
    if (! ("type" in obj)) {
        return false;
    }
    if (obj[type] !== "Book") {
        return false;
    }
    if (! ("title" in obj)) {
        return false;
    }
    if (! ("author" in obj)) {
        return false;
    }
    if ("publisher" in obj) {
        return false;
    }
    if ("content" in obj) {
        return false;
    }
    if ("year" in obj) {
        return false;
    }

    return true;
}
```

(b)

Fig. 5. TypeRelations metamodel and a simplified code for a specific version.

versions cannot be explicitly represented in class diagrams, but a new kind of representation is needed. Instead, it is possible to show the elements inferred for the different versions of a pair, for instance, the *Book* entity contains the *authors* attribute whose type is a collection of strings along with an aggregate to *Author*. To generate these class diagrams, we have taken advantage of the tooling provided by EMF/Ecore [14] to represent metamodels as UML class diagrams. This illustrates the benefits of representing models and metamodels uniformly.

Validation is often needed when dealing with NoSQL databases. For instance, a developer would want to assure that all the objects retrieved and stored by a given application conform to a given entity version. When developing a new version of an application, for example, object validators (a.k.a. schema predicates) could be created so that the programmer can check each object that transfers to and from the database. Another scenario could be removing a given version of objects. Validators allow characterizing objects to perform a filter operation on the database.

Figure 5(a) show the metamodel used to specify relations between versions of an entity type with respect to the JSON object structure (*TypeRelations*). These models are obtained via a model-to-model transformation from the NoSQL Schema, and then a model-to-text transformation generates the validator functions that check the entity version of a given JSON object. Figure 5(b) shows a simplified code to assert a given entity version. The same approach could be used to generate specialized queries for specific versions. The metamodel defines a type discrimination (*TypeDiff*) as a set of hints (*TypeHint*) that a given JSON object should fulfill to be considered of a given entity version. For example, it has to contain a given field (*HasField*), or a field with a value (*HasFieldWithValue*) (e.g. "type" should be "Book").

6 Related Work

The extraction of explicit schemas for JSON-based technologies and applications is gaining attention as JSON is emerging as a *lingua franca* for information

interchange. Web services and NoSQL systems are two examples of technologies for which some proposals have been presented. This research effort is related to the works published over the years on schema inference and schema versioning for semi-structured data, specially XML documents.

In [10], an algorithm to extract schemas from aggregate-oriented NoSQL databases is presented. This algorithm adapts strategies proposed for extracting XML DTDs to JSON documents. A JSON schema is obtained as output. The authors suggest some database utilities similar to those proposed in this work, such as validators and objects mapper classes. They focus on calculating statistics and finding outliers in the data. Our work differs from this approach in several essential aspects: (i) the algorithm identifies the required and optional properties, but object versions are not obtained; (ii) an schema involves only a type of objects, and reference and aggregation relations are not considered; (iii) they do not specify how to cope with huge amounts of data; and (iv) we obtain a model that conforms to a metamodel, instead of a JSON Schema.

The *JSON Schema* initiative [7] has recently emerged to provide standard specifications for describing JSON schemas. Although its adoption is still very limited, some tools (e.g. validators, schema generators, documentation generators) have evidenced the usefulness of having JSON schemas. The notion of NoSQL schema presented in our work is more expressive than the JSON schemas in the standard, since NoSQL schemas contain aggregate and reference relationships between entities, and also entity versions are represented.

Some tools able of discovering a schema from NoSQL databases have recently emerged. For instance, Spark SQL query engines [15] and Drill [2] are examples of such tools. In Spark SQL, a schema is described as a set of Scala algebraic types and can be inferred for a given set of JSON objects. Spark addresses object versions by means of "sum types", that is, creating types that contain all the keys in all the objects of an entity type, allowing them to be *null* in the objects created or received. As for conflicting types, it generalizes to a String type, that is able to represent any value. This may allow these conflicting types to be addressed without crashing, but it does not offer any guarantee regarding the consistency of the data. Instead, our approach discovers and represents the exact set of versions existing for each object type. Thus, versioned schemas are complete, and allow having a more fine grained control of the objects that enter to and are obtained from a database. Moreover, the reference and aggregation relations between entities are not made explicit in Spark SQL. Drill dynamically discovers the schema during the processing of a query, but it cannot cope with conflicting objects (those that do not comply with the schema). Also, the discovered schema is just used for the purposes of Drill, and cannot be reused by other applications.

MongoDB-Schema [12] is an early prototype of a tool whose purpose is to infer schemas from JSON objects and MongoDB collections. Given a set of objects of the same collection, the inference algorithm obtains an schema that is represented by a JSON document similar to the raw schemas in Sect. 4.1. Moreover, metadata is added to each field in the root and embedded objects in form of a key/value pair. For instance "type" indicates the object type (e.g. Number,

String, or Boolean) and "count" indicates the number of objects that contains a field. Note that this approach has the same limitations of Spark SQL.

A MDE-based approach to infer JSON schemas from REST web services is proposed in [5]. A three-step process is performed to discover the domain model of the services. Firstly, the JSON data for a service is injected into models which conforms to a metamodel similar to that used here. In the second step, a mapping between the JSON metamodel and the Ecore meta-metamodel is established in order to transform the JSON model into a domain model. This JSON-to-Ecore mapping is similar to the one applied here for obtaining a visual representation of NoSQL schemas. In our case, Schema models are represented as Ecore metamodels, which is a more direct mapping. Finally, the domain models obtained for each service are integrated by superposing the common classes. This work is clearly close to our approach but there are however some significant differences between them, namely JSON Discoverer does not tackle the existence of data versions, and the references between objects are not discovered.

In [8] a strategy to infer schemas from heterogeneous XML databases is presented. The schema is provided as a Schema Extended Context-Free Grammar, and the different versions are integrated into a single grammar which is mapped to a relational database schema. In our case, we have used a metamodel to represent the schemas, which has allowed us to apply MDE techniques, and we keep the different versions instead of obtaining a single schema. As in previous related works, our schema is richer, taking into account aggregations and references.

Finally, a design method for aggregate-based NoSQL database is proposed in [3]. This method defines the NoAM model to represent these databases in a system-independent way. NoAM is similar to our Schema metamodel, but it does not consider the possible existence of database object versions. Moreover, the model is simply proposed but it is not implemented in form of a metamodel.

7 Conclusions and Future Work

To bring the well-known benefits of schemas to NoSQL databases, several approaches have been proposed to infer the schema from the data. However, these tools do not cope well with the variability of the schemaless data: they either do not support variation in the structure of the objects of a given type, or they overgeneralize the schema to embrace all the possible variations. In our proposal, the schema takes into account the existing versions of each type: the Versioned Schema has the unique characteristic of completely defining the structure of the data, also showing the high-level relationships, such as aggregation and reference.

The presented approach has proved useful in generating specifications that describe the data, as well as in generating useful applications within the development technical space of the NoSQL databases, such as type validators.

After this initial effort, future directions include generating data visualizations that take into account the type *and* version of the objects in the database, allowing to visually identify the quantities of objects of each type and version.

If a data base has evolved over time, it would be interesting to show which data belong to each version.

Generating object version transformers could also be interesting. A developer can describe, by means of a specialized DSL, the necessary steps to convert one version of an object to another version. These could be used in at least two ways:

- A new application that uses the stored old data may require that all the recovered objects comply with the new version. A version transformer could be generated that removes the unneeded fields, and gives values to new, non-existing fields. This would guarantee that the application would always use object with the correct (new) version, giving all the process more robustness.
- Batch database migration. Map-Reduce jobs could be generated to transform old version objects into new versions. This is possible given the precise version information stored in the schema.

References

1. Abiteboul, S.: Querying semi-structured data. Technical report 1996–19, Stanford InfoLab (1996). http://ilpubs.stanford.edu:8090/144/
2. Apache Foundation: Apache Drill, Visited April 2015. http://drill.apache.org/
3. Bugiotti, F., Cabibbo, L., Atzeni, P., Torlone, R.: Database design for NoSQL systems. In: Yu, E., Dobbie, G., Jarke, M., Purao, S. (eds.) ER 2014. LNCS, vol. 8824, pp. 223–231. Springer, Heidelberg (2014)
4. Buneman, P.: Semistructured data. In: Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 117–121. ACM (1997)
5. Cánovas Izquierdo, J.L., Cabot, J.: Discovering implicit schemas in JSON data. In: Daniel, F., Dolog, P., Li, Q. (eds.) ICWE 2013. LNCS, vol. 7977, pp. 68–83. Springer, Heidelberg (2013)
6. Fowler, M.: Schemaless Data Structures, January 2013. http://martinfowler.com/articles/schemaless/
7. IETF: JSON Schema Specification, Visited April 2015. http://json-schema.org/
8. Janga, P., Davis, K.C.: Mapping heterogeneous XML document collections to relational databases. In: Yu, E., Dobbie, G., Jarke, M., Purao, S. (eds.) ER 2014. LNCS, vol. 8824, pp. 86–99. Springer, Heidelberg (2014)
9. Karpov, V.: Mongoose NPM package, Visited April 2015. https://www.npmjs.com/package/mongoose
10. Klettke, M., Störl, U., Scherzinger, S.: Schema extraction and structural outlier detection for JSON-based NoSQL data stores. In: BTW 2105, pp. 425–444 (2015)
11. Redmond, E., Wilson, J.R.: Seven Databases in Seven Weeks. A Guide to Modern Databases and the NoSQL Movement, Pragmatic Programmers (2013)
12. Rückstieß, T.: mongodb-schema NPM package, Visited April 2015. https://www.npmjs.com/package/mongodb-schema
13. Sadalage, P., Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, Reading (2012)
14. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework. Addison-Wesley, Reading (2008)
15. Zaharia, M., Chowdhury, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI, April 2012. http://spark.apache.org