Reverse engineering process for NoSQL databases

***Abstract*—NoSQL systems have proven their efficiency to handle Big Data. Most of these systems are schema-less which means that the database doesn't have a fixed data structure. This property offers an undeniable flexibility allowing the user to add new data without making any changes on the data model. However, the lack of an explicit data model makes it difficult to express queries on the database. Therefore, users (developers and decision-makers) still need the database data model to know how data are stored and related, and then to write their queries. In previous work, we have proposed a process to extract the physical model of a document-oriented NoSQL database. In this paper, we aim to extend this work to achieve a reverse engineering of NoSQL databases in order to provide an element of semantic knowledge close to human understanding. The reverse engineering process is ensured by a set of transformation algorithms. We provide experiments of our approach using a case study example taken from the health field. We also propose a validation of our solution in a real context; the results of this validation show that the generated conceptual model provide a good assistance to the user to express their queries on the database while saving a lot of time.**

Keywords—Reverse Engineering, NoSQL, Big Data, Schema-less, Conceptual Model.

I. Introduction

Big Data have attracted a great deal of attention in recent years thanks to the huge amount of data managed, the types of data supported and the speed at which this data is collected and analyzed. This has definitely impacted the tools required to store Big Data, and new kinds of data management tools i.e. NoSQL systems [7]. Compared to existing DBMSs, NoSQL systems are generally known for handling a large amount of data, undeniable flexibility and faster access to data. [1].

One of the NoSQL key features is that databases can be schema-less. This means, in a table, meanwhile the row is inserted, the attributes names and types are specified. This property offers an undeniable flexibility that facilitates the data model evolution and allows end-users to add new information without the need of database administrator; but, at the same time, it makes the database manipulation more difficult. Indeed, even in Big Data context, the user still needs a data model that offers a visibility of how data is structured in the database (tables name, attributes names and types, relationships, etc.). In practice, the developer that has created the database, is also in charge of writing queries. Thus, he already knows how data is stored and related in the database; so, he can easily express his requests. However, this solution cannot be applied to all cases; for instance, the developer who is asked for maintaining the application, does not know the data model. It is the same for a decision maker who needs to query a database while he was not involved in its creation.

On the one hand, NoSQL systems have proven their efficiency to handle Big Data. On the other hand, the needs of a NoSQL database model remain up-to-date. Therefore, we are convinced that it's important to provide to users (developers and decision-makers) two data models describing the database: (1) a physical model that describes the internal organization of data and allows to express queries and (2) a conceptual model that provides a high level of abstraction and a semantic knowledge element close to human comprehension, which guarantees efficient data management [20].

In a previous work, we have proposed a process for extracting a physical model starting from a NoSQL database. In this paper, we aim to propose an extension of this work by transforming the physical model (already obtained) into a conceptual model. A reverse engineering process will be used for this ; it is ensured by the application of a set of algorithms

The remainder of the paper is structured as follows. Section 2 motivates our work using an example in the healthcare field. Section 3 reviews previous work. Section 4 describes our reverse engineering process. Section 5 details our experiments and compare our solution against those presented in Section 3. Section 6 validates our solution. Finally, Section 7 concludes the paper and announces future work.

II. Illustrative example

To motivate and illustrate our work, we have used a case study in the healthcare field. This case study concerns international scientific programs for monitoring patients suffering from serious diseases. The main goal of this program is (1) to collect data about diseases development over time, (2) to study interactions between different diseases and (3) to evaluate the short and medium-term effects of their treatments. The medical program can last up to 3 years. Data collected from establishments involved in this kind of program have the features of Big Data (the 3 V): Volume: the amount of data collected from all the establishments in three years can reach several terabytes. Variety: data created while monitoring patients come in different types; it could be (1) structured as the patient's vital signs (respiratory rate, blood pressure, etc.), (2) semi-structured document such as the package leaflets of medicinal products, (3) unstructured such as consultation summaries, paper prescriptions and radiology reports. Velocity: some data are produced in continuous way by sensors; it needs a [near] real time process because it could be integrated into a time-sensitive processes (for example, some measurements, like temperature, require an emergency medical treatment if they cross a given threshold).

In these programs, one of the benefits of using NoSQL databases is that the evolution of the data (and the model) is fluent. In order to follow the evolution of the pathology, information is entered regularly for a cohort of patients. But the situation of a patient can evolve rapidly which needs the recording of new information. Thus, few months later, each patient will have his own information, and that's how data will evolve over time. Therefore, the data model (1) differs from one patient to another and (2) evolves in unpredictable way over time.

As mentioned before, this kind of systems operate on schema-less data model enabling developers to quickly and easily incorporate new data into their applications without rewriting tables. Nevertheless, there is still a need for a conceptual model to know how data is structured and related in the database; this is particularly necessary to write declarative queries where tables and columns names are specified [4].

In our view, it's important to have a precise and automatic solution that guides and facilitates the database model extraction task within NoSQL systems. For this, we propose the *ToConceptualModel* process presented in section 4 that extracts a conceptual model of a NoSQL database. This model is expressed using a UML class diagram.

III. Related work

The problem of extracting the data model from schema-less NoSQL databases has been the subject of several research works. Most of these works focus on the physical level [17], [20], [11], [18], [2], [3] and [10]. In this context, we have proposed a process to extract a document-oriented database physical model. This process applies a sequence of transformations formalized with the QVT standard proposed by the Object Management Group (OMG). Which is original in our solution is that it takes into account the links between different collections.

However, we should highlight that only few works, [8], [15] and [21], have addressed the extraction of a NoSQL database conceptual model. In [8], the authors propose an extraction process of a conceptual model for a graph-oriented NoSQL database (Neo4J). In this particular type of NoSQL databases, the database contains nodes (objects) and binary links between them. The proposed process takes as input the insertion requests of objects and links; and then returns an Entity / Association model. This process is based on an MDA architecture and successively applies two transformations. The first is to build a graph (Nodes + Edges) from the Neo4j query code. The second consists of extracting an Entity / Association model from this graph by transforming the nodes with the same label into entities and the edges into associations. These works are specific to graph-oriented NoSQL databases generally used to manage strongly linked data such as those from social networks.

Authors in [15] propose a process to extract a conceptual model (UML class diagram) from a JSON document. This process consists of 2 steps. The first step extracts a physical model in JSON format. The second step generates the UML class diagram by transforming the physical model into a root class RC, the primitive fields (Number, String and Boolean) into attributes of RC and the structured fields into component classes linked to RC by composition links. Thus, this work only considers the composition links and ignores other kinds of links (association and aggregation links for example).

The process proposed in [21] deals with the mapping of a document-oriented database into a conceptual model. It consists of a set of entities with one or more versions. This work considers the two types of links: binary-association and composition. Binary-association and composition links are respectively extracted using the reference and structured fields. This solution does not consider other links that are usually used like n-ary, generalization and aggregation links.

In Table 1, we summarize the three previous works according to three factors: modeling levels (Physical and/or conceptual), NoSQL system type and links types.

Regarding the state of the art, the existing solutions have the advantage to start from the conceptual level, but they do not consider all the UML class diagram features that we need for our medical use case. Indeed, the process in [8] concern only graph-oriented systems that, unlike document-oriented databases, do not allow to express structured attributes and composition links. On the other hand, the solution of [15] starts from a document-oriented database but do not consider aggregation, generalization and also association links that are used to link data in our case study. As [15], authors in [21] use a document-oriented database, but do not take into account the generalization and aggregation links, association classes and also n-ary association links that are the most used in the medical application.

The main purpose of our work is to complete these solutions. For this, we have proposed an automatic approach that transforms a document-oriented physical model into a UML class diagram. This automatic process takes into account several kinds of links: association classes, binary and n-ary association links, and also the generalization, composition and aggregation links.

TABLE I. Comparative table of extraction works of conceptual model from schema-less NoSQL databases

	Modeling Level		NoSQL System Type		Link Types		
	Physical	Conceptual	Graph	Document	Binary association	Composition	Generalization
[8]	X	X	X		X		X
[15]	X	X		X		X	
[21]	X	X		X	X	X	

IV. Reverse engineering process

Our work aims to provide users with models to manipulate NoSQL databases. Two models are proposed: (1) the physical model to write queries on this database and application code and (2) the conceptual model to give the meaning of the data contained in the database. When data structures are complex, these two models are essential to enable users (usually, developers and decision-makers) to understand and manipulate data independently.

As part of this work, we proposed mechanisms for discovering a physical model from a NoSQL database in a previous paper. The current paper completes the latter and focuses on the transformation of the physical model into a conceptual model represented by using a UML class diagrams (red frame in Figure 1) and which provides users with the semantics of the data.

Note that we limit our study to document-oriented NoSQL databases that are the most complete to express links between objects (use of referenced and nested data).

We propose the *ToConceptualModel* process which applies a set of transformations ensuring the passage of a NoSQL physical model towards a UML class diagram.

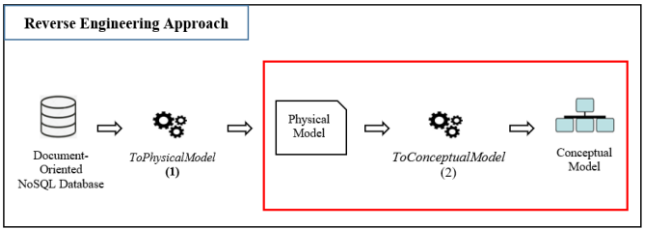


Fig. 1. Overview of ToConceptualModel process

In the following sections, we detail the components of the *ToConceptualModel* process by specifying the three elements: (a) the source, (b) the target and (c) the transformation algorithms.

A. Source: Physical model

The physical model is produced by the *ToPhysicalModel* process (shown in Figure 1). In this paper, it is the source of the *ToConceptualModel* process that we will study here. The physical model is defined as a pair (N, CL), where:

- N is the physical model name,
- $CL = \{cl_1, \dots, cl_n\}$ is a set of collections. ∀ i ∈ [1..n], a collection $cl_i \in$ CL is defined as a pair (N, F), where:
  - $cl_i.N$ is the collection name,
  - $cl_i.F = AF \cup SF$ is a set of fields, where:
    - $AF = \{af_1, \dots, af_m\}$ is a set of atomic fields. ∀ j ∈ [1..m], an atomic field $af_j \in AF$ is defined as a tuple (N, T, M), where:
      - af_j.N is the af_j name,
      - af_j.T is the af_j type; it is one of the standard data types such as Integer, String, Boolean, …,
      - af_j.M is a boolean which indicates whether af_j is multivalued or not.
    - $SF = \{sf_1, \dots, sf_l\}$ is a set of structured fields. ∀ k ∈ [1..l], a structured field $sf_k \in SF$ is defined as a tuple (N, F', M), where:
      - sf_k.N is the sf_k name,
      - $sf_k.F' = AF' \cup SF'$ is a set of fields that compose the structured field sf_k (see above),
      - sf_k.M is a boolean which indicates whether sf_k is multivalued or not.

To express a link between two collections, we used a field called DBRef, which is un standard proposed by MongoDB [19]. A DBRef field is a special case of a structured field (N, F', M), where: N is the link name; F' contains two atomic fields: ***$id: ObjectId*** which corresponds to the identifier of the referenced document and ***$Ref: String*** which corresponds to the name of the collection that contains the referenced document; M indicates whether the link is monovalued or multivalued.

We have extended the DBRef syntax to take into account n-ary links and association classes. In this new syntax, F' can contain several pairs ***($id: ObjectId, $Ref: String)*** and possibly other fields. To create a generalization link between collections, we propose using a DBSub field in the sub-collection. DBSub is a special case of structured field where N = Sub; F' contains two atomic fields: ***$id: ObjectId*** which identifies the generic document and ***$Sub: String*** corresponds to the super-collection name; M = 0. To express an aggregation link between collections, we suggest using a DBAgg field in the aggregate collection. DBAgg is a special case of structured field (N, F', M) where N = Agg; F' contains two atomic fields: ***$id: ObjectId*** which identifies the part document (aggregated) and ***$Agg: String*** corresponds to the name of the part collection.

B. Target: conceptual model

A UML Class Diagram (CD) is defined as a tuple (N, C, L), where:

- N is the CD name,
- C = $\{c_1, \dots, c_n\}$ is a set of classes,
- L = $AL \cup CL \cup AGL \cup GL$ is a set of links

1) Classes:

∀ i ∈ [1..n], a class $c_i \in$ C is defined as a pair (N, A), where:

- $c_i.N$ is the class name,
- $c_i.A = AA \cup SA$ is a set of attributes, where:
  - $AA = \{aa_1, \dots, aa_m\}$ is a set of atomic attributes. ∀ j ∈ [1..m], an atomic attribute $aa_j \in AA$ is defined as a tuple (N, T, M), where:
    - aa_j.N is the aa_j name,
    - aa_j.T is the aa_j type; T can have the value: String, Integer, Boolean…,
    - aa_j.M is a boolean which indicates whether aa_j is multivalued or not.
  - $SA = \{sa_1, \dots, sa_l\}$ is a set of structured attributes. ∀ k ∈ [1..l], a structured attribute $sa_k \in SA$ is defined as a tuple (N, A', M), where:
    - sa_k.N is the sa_k name,
    - $sa_k.A' = AA' \cup SA'$ is a set of attributes that compose sa_k (see above),
    - sa_k.M is a boolean which indicates whether sa_k is multivalued or not.

2) Links:

- $AL = \{al_1, \dots, al_m\}$ is a set of association links. ∀ i ∈ [1..m], an association link $al_i \in AL$ is defined as a tuple (N, RC, A), where:
  - $la_i.N$ is the al_i name,
  - $la_i.RC = \{rc_1, \dots, rc_f\}$ a set of related collections with degree f ≥ 2. ∀ j ∈ [1..f], rc_j is defined as a pair (c, cr), where:
    - rc_j.c is the related class name,
    - $rc_j.cr$ is the multiplicity corresponding to c.
  - $la_i.A = AA \cup SA$ is a set of attributes of al_i (see above). Note that if $al_i.A \neq \emptyset$ then al_i is an association class.
- $CL = \{cl_1, \dots, cl_m\}$ is a set of composition links. ∀ i ∈ [1..m], a composition link $cl_i \in CL$ is defined as a pair $(rc^{composite}, rc^{component})$ où :
  - $cl_i.rc^{composite}$ is a pair defining the composite class; it is in the form of (c, cr), where:
    - $rc^{composite}$.c is the composite class name.
    - $rc^{composite}.cr$ is the multiplicity corresponding to the composite class. This multiplicity generally contains the value 0..1, 1..1 or 1 for the contracted form.

- $cl_i.rc^{component}$ is a pair defining the component class; it is in the form of (c, cr), where:
  - $rc^{component}$.c is the component class name.
  - $rc^{component}.cr$ is the multiplicity corresponding to the component class.

- $AGL = \{agl_1, \dots, agl_m\}$ is a set of aggregation links. ∀ i ∈ [1..m], an aggregation link $agl_i \in AGL$ is defined as a pair $(rc^{aggregate}, rc^{part})$, where:
  - $agl_i.rc^{aggregate}$ is a pair defining the aggregate class; it is in the form of (c, cr), where:
    - $rc^{aggregate}$.c is the aggregate class name,
    - $rc^{aggregate}.cr$ is the multiplicity corresponding to the aggregate class,
  - $agl_i.rc^{part}$ is a pair defining the part class; it is in the form of (c, cr), where:
    - rc^{part}.c is the part class name,
    - $rc^{part}.cr$ is the multiplicity corresponding to the part class,
- $GL = \{gl_1, \dots, gl_m\}$ is a set of generalization links. ∀ i ∈ [1..m], a generalization link $gl_i \in LH$ is defined as a pair (sc, SSC), where:
  - gl_i.sc is the super-class name,
  - $gl_i.SBC = \{sbc_1, \dots, sbc_k\}$, where: ∀ j ∈ [1..k], with k ≥ 1, ssc_j is a sub-class.

C. Transformation algorithms:

The mapping from the physical model to the conceptual model is ensured by applying six transformation algorithms: $\boldsymbol{TA_{Collection}}$, $\boldsymbol{TA_{AtomicField}}$, $\boldsymbol{TA_{StructuredField}}$, $\boldsymbol{TA_{DBRef}}$, $\boldsymbol{TA_{DBSub}}$ and $\boldsymbol{TA_{DBAgg}}$.

➢ $\boldsymbol{TA_{Collection}}$

This algorithm transforms a collection into a class. $TA_{Collection}$ possesses the following properties:

Input:

$cl = (N, F)$: a collection defined by a name N and a set of fields $F = AF \cup SF$.

Output:

$c = (N, A)$: a class defined by a name N and a set of attributes $A = AA \cup SA$.

Definition:

$\mathbf{TA_{Collection}}$

In: cl
Out: c
1 Begin
2 c.N = cl.N
3 $c.A$ is generated by applying algorithms relating to the transformation of the collection fields.
4 $C = C \cup c$ *// Add c to the set classes C*
5 End

➢ $\boldsymbol{TA_{AtomicField}}$

This algorithm transforms an atomic field into an atomic attribute. $TA_{AtomicField}$ possesses the following properties:

Input:

$af = (N, T, M)$: an atomic field defined by a name N, a type T and a boolean M.

Output:

$aa = (N, T, M)$: an atomic attribute defined by a name N, a type T and a boolean M.

Definition:

$\mathbf{TA_{AtomicField}}$

In: af
Out: aa
1 Begin
2 aa.N = af.N
3 aa.T = af.T
4 aa.M = af.M
5 End

➢ $\boldsymbol{TA_{StructuredField}}$

This algorithm transforms a structured field which is not a DBRef. The result of this transformation can be either a composition link if the structured field consists of at least one DBRef field or a structured attribute otherwise. $TA_{StructuredField}$ possesses the following properties:

Input:

$sf = (N, F', M)$: a structured field defined by a name N, a set of fields $F' = AF' \cup SF'$ and a boolean $M.sf$ is declared in the collection c_0.

Output:

$cl = (rc^{composite}, rc^{component})$: a composition link defined by a composite class $rc^{composite}$ and a component class $rc^{component}$.
Or $sa = (N, A', M)$: a structured attribute defined by a name N, a set of attributes A' and a boolean M.

Definition:

$\mathbf{TA_{StructuredField}}$

In: sf
Out: cl or sa
1 Begin
2 **If** ∃ $dbref \in sf.SF'$ **Then** *// This is a composition link*
3 $cl.rc^{composite}.c = c_0$.N
4 $cl.rc^{composite}.cr = 1..1$
5 $cl.rc^{component}.c = sf$.N
6 **If** $sf.M = 0$ **Then** *// If sf is monovalued*
7 $cl.rc^{component}.cr = 0..1$
8 **Else** *// If sf is mulivalued*
9 $cl.rc^{component}.cr = 0..*$
10 **End If**
11 Get al by applying $\mathbf{TA_{DBRef}}$ on $dbref$ *// Create an association link between the component class and the class referenced in dbref*
12 **Else** *// This is a structured attribute*
13 sa.N = sf.N
14 **For** every $af_i \in sf.AF'$, with i ∈ [1..m] **do**
15 Get aa_i by applying $\mathbf{TA_{AtomicField}}$
16 $sa.AA' = sa.AA' \cup aa_i$
17 **End For**

18 **For** every $sf_j \in sf.SF'$, with j ∈ [1..l] **do**
19 Get sa_j by applying $\mathbf{TA}_{StructuredField}$
20 $sa.SA' = sa.SA' \cup sa_j$
21 **End For**
22 sa.M = sf.M
23 **End If**
24 End

➢ TA_{DBRef}

This algorithm transforms a DBRef field into an association link. TA_{DBRef} possesses the following properties:

Input:

$dbref$ = (N, F', M): a DBRef field defined by a name N, a set of fields F' (composed of *n* pairs ***($id: ObjectId, $Ref: C_i)*** with i ∈ [1..n] and possibly, *m* atomic fields and *l* structured fields) and a boolean M. $dbref$ is declared in the collection c_0.

Output:

al = (N, RC, A): an association link defined by a name N, a set of related classes RC and a set of attributes $A = AA \cup SA$.

Definition:

$\mathbf{TA}_{DBRef}$

In: $dbref$
Out: al
1 Begin
2 $al.RC = \emptyset$
3 $al.A = \emptyset$
4 al.N = $dbref$.N
5 **If** $n = 1$ **Then** *// It is a binary link*
6 rc_0.c = c_0.N
7 rc_1.c = c_1.N
8 rc_0.cr = *0..**
9 **If** $dbref$.M = 0 **Then** *// if dbref is monovalued*
10 rc_1.cr = *0..1*
11 **else** *// if dbref is multivalued*
12 rc_1.cr = *0..**
13 **End If**
14 $al.RC = rc_0 \cup rc_1$
15 **Else** *// $n > 1$ (It is an n-ary link)*
16 rc_0.c = c_0.N
17 **If** $dbref$.M = 0 **Then** *// if dbref is monovalued*
18 rc_0.cr = *0..1*
19 **Else** *// if dbref is multivalued*
20 rc_0.cr = *0..**
21 **End If**
22 $al.RC = rc_0$
23 **For** i ∈ [1..n] **do**
24 rc_i.c = c_i.N
25 rc_i.cr = *0..**
26 $al.RC = al.RC \cup rc_i$
27 **End For**
28 **End If**
29 **For** every $af_j \in dbref.AF'$, with j ∈ [1..m] **do** *// Extract the atomic attributes of the association class*
30 Get aa_j by applying $\mathbf{TA}_{AtomicField}$
31 $al.AA = al.AA \cup aa_j$
32 **End For**
33 **For** every $sf_k \in dbref.SF'$, with k ∈ [1..l] **do** *// Extract the structured attributes of the association class*
34 Get sa_k by applying $\mathbf{TA}_{StructuredField}$
35 $al.SA = al.SA \cup sa_k$
36 **End For**
37 $al.A = al.AA \cup al.SA$
38 End

Example:

In the example illustrated in Figure 2, the association link Treating-Doctors between Patients and Doctors is obtained by applying TA_{DBRef}.

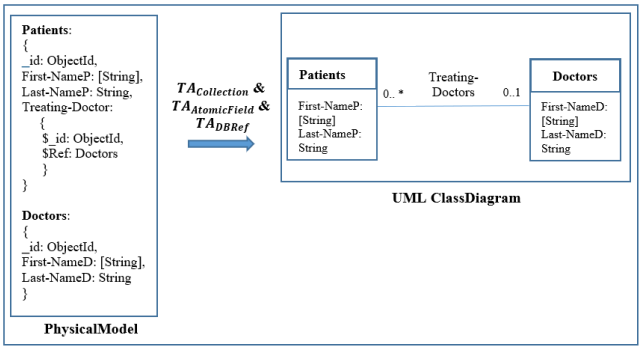


Fig. 2. Example of extracting an association link

➢ TA_{DBSub}

This algorithm transforms a DBSub field into a generalization link. TA_{DBSub} possesses the following properties:

Input:

$dbsub$ = (N, F', M): a DBSub field defined as a structured field whose N = Sub; F' consists of two atomic fields: ***$id: ObjectId*** and ***$Sub: SCl***; the value of M is 0. $dbsub$ is declared in the collection SbCl.

Output:

gl = (sc, SBC): a generalization link defined by a super-class sc and a set of sub-classes SBC.

Definition:

$\mathbf{TA}_{DBSub}$

In: $dbsub$
Out: gl
1 Begin
2 gl.sc = SCl
3 gl.SBC = gl.SBC ∪ $SbCl$
4 End

➢ TA_{DBAgg}

This algorithm transforms a DBAgg field into an aggregation link. TA_{DBAgg} possesses the following properties:

Input:

$dbagg$ = (N, F', M) : a DBAgg field defined by N = Agg, a set of fields F' (consists of two atomic fields ***$id: ObjectId*** and ***$Agg: c_1)*** and a boolean M = 0. $dbagg$ is declared in the part collection c_0.

Output:

$agl = (rc^{aggregate}, rc^{part})$: an aggregation link defined by an aggregate class $rc^{aggregate}$ and a part class rc^{part}.

Definition:

TA$_{DBAgg}$

In: $dbagg$
Out: agl
1 Begin
3 $agl.rc^{aggregate}.c = c_1.N$
4 $agl.rc^{part}.c = c_0.N$
5 $agl.rc^{part}.cr = 0..*$
6 **If** $dbagg.M = 0$ **Then** *// if dbagg is monovalued*
7 $agl.rc^{aggregate}.cr = 0..1$
8 **Else** *//if dbagg is multivalued*
9 $agl.rc^{aggregate}.cr = 0..*$
10 **End If**
14 **End**

V. Experiments and Comparison:

A. Technical environment

In this section, we describe the techniques used to implement the *ToConceptualModel* process. We used a technical environment suitable for modeling, meta-modeling and model transformation. We used the Eclipse Modeling Framework (EMF) [6]. EMF provides a set of tools for introducing a model-driven development approach within the Eclipse environment. These tools provide three main features. The first is the definition of a meta-model representing the concepts handled by the user. The second is the creation of the models instantiating this meta-model and the third one is the transformation from model to model and from model to text. Among the tools provided by EMF, we used:

- Ecore: a metamodeling language used to create our metamodels used by *ToConceptualModel* process.
- XML Metadata Interchange (XMI): the XML based standard that we use to create models.
- QVT (Query, View, Transformation): the OMG standard language for specifying model transformations. The choice of the QVT standard was based on criteria specific to our approach. Indeed, the transformation tool must be integrated into the EMF environment so that it can be easily used with modeling and meta-modeling tools.

B. Implantation of the ToConceptualModel process

ToConceptualModel process is expressed as a sequence of elementary steps that build the resulting model (UML class diagram) step by step from the source model (physical model).

Step1: we create a source and a target metamodel to represent the concepts handled by our process.

Step2: we build an instance of the source metamodel. For this, we use the standard based XML Metadata Interchange (XMI) format. This instance is shown in Figure 3.

Step3: we implement the transformation algorithms by means of the QVT language provided within EMF.

Step4: we test the transformation by running the QVT script created in step 3. This script takes as input the source model built in step 2 (physical model) and returns as output a UML class diagram. The result is provided in the form of XMI file as shown in figure 4.

C. Comparison

The aim of this section is to compare our solution with the three works [8], [15], [21] presented in section 3 and that have investigated the process of extracting a NoSQL database conceptual model. Starting from a graph-oriented NoSQL database, authors in [8] propose to extract an E/A model based on a set of mapping rules between the conceptual level and the physical one. Obviously, these rules are specific to graph-oriented systems used as a framework for managing complex data with many connections. This kind of NoSQL DBMS lack of ability to define structured attributes, association classes as well as n-ary, composition and aggregation links that we need to use in our use case (cf. Section 2). The solution presented in [15] have the advantage to start from a document-oriented NoSQL database. The only type of links considered in this work is the composition link; other types are not taken into account. Other process in [21] focuses on binary association and composition links during the extraction of a document-oriented NoSQL database. However, it doesn't consider structured attributes, association classes as well as n-ary, generalization and aggregation links.

To overcome the limits of these works, we have proposed a more complete solution which addresses different types of attributes and links: atomic and structured attributes, association (binary and n-ary), generalization, aggregation and composition links as well as association classes.

VI. Validation

Concerning the model extraction of schema-less NoSQL databases, our approach allows to display to the developer simultaneously a physical model and a conceptual model; the first to write queries and the second to understand the semantics of the database. To evaluate the relevance of our approach, our prototype (section 4) was implemented by three developers at Trimane, a digital services company specialized in business intelligence and Big Data.

The three experienced developers (IT consulting engineers) were tasked with providing maintenance for three separate applications. None of the developers know, previously, the data model of the concerned applications. For each application, each developer writes ten queries that have an increasing complexity according to three different cases: (1) without any data model, (2) with the physical data model or (3) with the both conceptual and physical models. Figures 5 (a) and 5 (b) show respectively an example of the Physical and conceptual models corresponding to one of the three applications. Note that due to lack of place, we present data models (conceptual and physical one) of only one application.

We should also highlight that for reasons of visibility, models are represented to the user in the same screen and with an appropriate format (JSON for the physical model and the graphic format for the conceptual one); moreover, we do not show the attributes of classes. Each time we click on a collection on the physical model, we will have its equivalent on the conceptual model. For example, the gray class in the conceptual model corresponds to the part written in bold in the physical model (Trials).

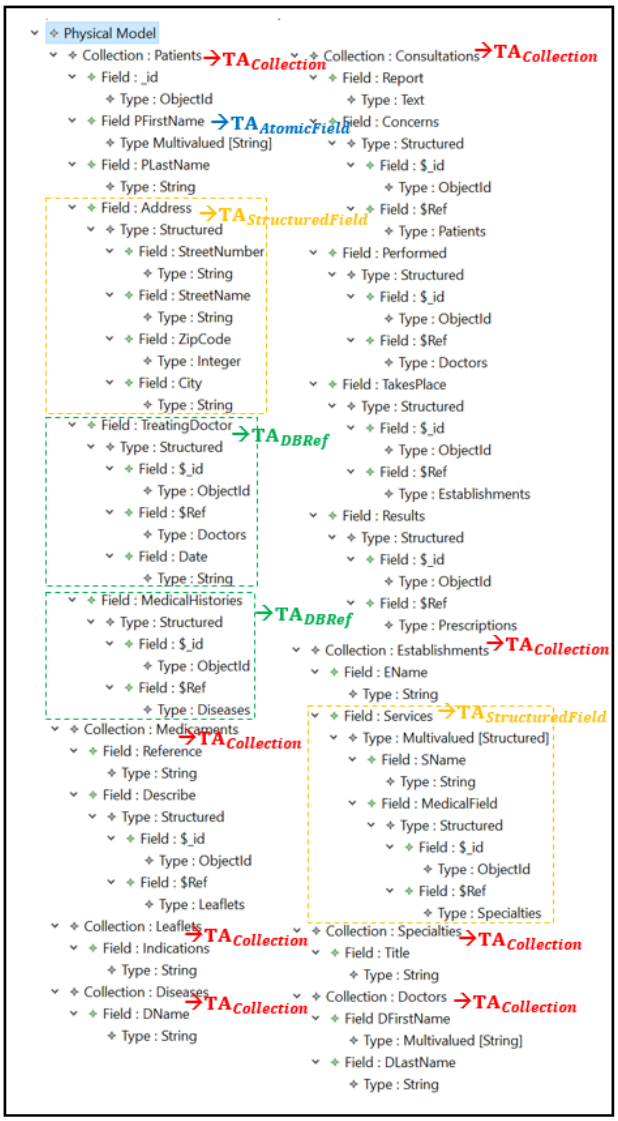


Fig. 3. Source Model

Fig. 4. Target Model

Each database is associated with a set of queries whose natural language statements are provided to the three developers. In Table 2, we calculated the average time of writing the queries by the three developers in each situation: (1) without any data model, (2) with the physical data model or (3) with the both conceptual and physical models.

TABLE II. QUERY WRITING TIME

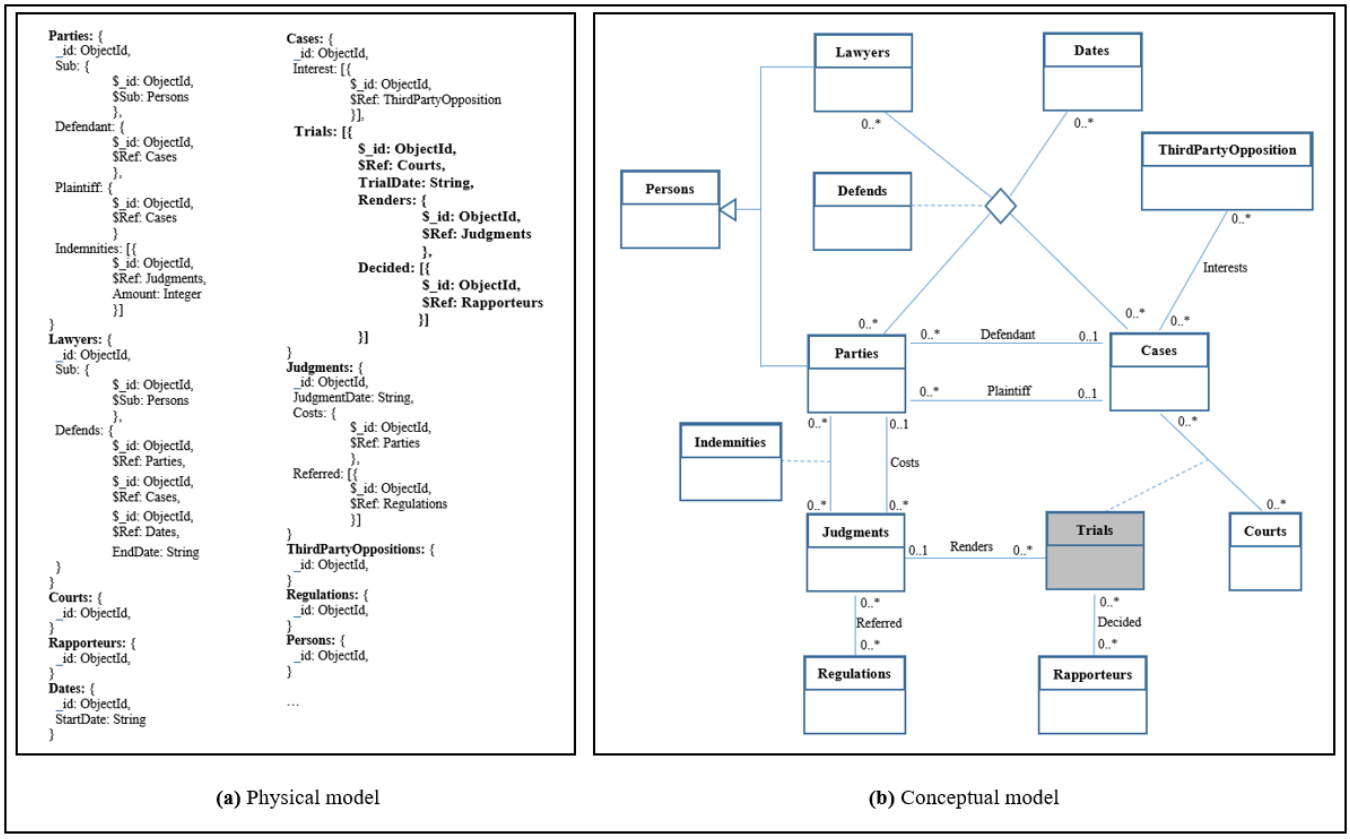
	Without model	**Physical model alone**	**Conceptual and physical model**
Developer 1	Database 1: 50 minutes	Database 2: 23 minutes	Database 3: 18 minutes
Developer 2	Database 2: 40 minutes	Database 3: 25 minutes	Database 1: 16 minutes
Developer 3	Database 3: 48 minutes	Database 1: 20 minutes	Database 2: 16 minutes
Average	**46 minutes**	**23 minutes**	**17 minutes**

Our initial hypothesis was verified in the situations considered. This establishes that a knowledge of semantics and data structure allows the developer to write queries faster on a schema-less NoSQL database. The small difference noted between the use of the single physical diagram and the use of the two models (conceptual and physical), is probably due to the experience of the three developers.

VII. CONCLUSION AND FUTURE WORK

Our works are dealing with the reverse engineering mechanisms of schema-less NoSQL databases ; the purpose is to provide users with models to manipulate this type of database. We have presented in this paper an automatic process for mapping a UML conceptual model starting from a NoSQL physical model. We note that we have proposed, in previous work, a process to extract a NoSQL physical model starting from a document-oriented NoSQL database. So, we use this physical model to generate a conceptual model that makes it easier for developers and decision-makers to (1) understand how data are stored and related in the database and (2) write their queries. The mapping between the two models, physical and conceptual, is ensured by a set of transformation algorithms.

Regarding the state of the art, which is original in our solution is that it takes into account different types of attributes and links: atomic and structured attributes, association (binary



(a) Physical model **(b)** Conceptual model

Fig. 5. Screen representing the database of one of the three applications

and n-ary), generalization, aggregation and composition links as well as association classes.

We have experimented our process using a case study in the health field. We have also validated our solution in three different real cases to prove that the generated conceptual model provides a good assistance to the user to express their queries on the database while saving a lot of time. As future work, we plan to complete our transformation process to have more semantics in the conceptual model by taking into account other types of links such as reference links.

REFERENCES

[1] Angadi, A. B., & Gull, K. C. (2013). Growth of New Databases & Analysis of NOSQL Datastores. International Journal of Advanced Research in Computer Science and Software Engineering, 3, 1307-1319.

[2] Baazizi, M. A., Lahmar, H. B., Colazzo, D., Ghelli, G., & Sartiani, C. (2017, March). Schema inference for massive JSON datasets. In Extending Database Technology (EDBT).

[3] Baazizi, M. A., Colazzo, D., Ghelli, G., & Sartiani, C. (2019). Parametric schema inference for massive JSON datasets. The VLDB Journal, 1-25.

[4] Bondiombouy, C. (2015). Query processing in cloud multistore systems. In BDA : Bases de Données Avancées.

[6] Budinsky, F., Steinberg, D., Ellersick, R., Grose, T. J., & Merks, E. (2004). Eclipse modeling framework: a developer's guide. Addison-Wesley Professional.

[7] Chen, CL Philip et Zhang, Chun-Yang. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. Information Sciences, 2014, vol. 275, p. 314-347.

[8] Comyn-Wattiau, I., & Akoka, J. (2017, December). Model driven reverse engineering of NoSQL property graph databases: The case of Neo4j. In 2017 IEEE International Conference on Big Data (Big Data) (pp. 453-458). IEEE.

[9] Douglas, L., 2001. 3d data management: Controlling data volume, velocity and variety. Gartner. Retrieved, 6, 2001.

[10] Extract Mongo Schema https://www.npmjs.com/package/extract-mongo-schema/v/0.2.9 Online; 5 October 2019.

[11] Gallinucci, E., Golfarelli, M., & Rizzi, S. (2018). Schema profiling of document-oriented databases. Information Systems, 75, 13-25.

[12] Generate Test Data (2018) http://www.convertcsv.com/generate-test-data.htm Online; 5 July 2018.

[13] Han, Jing, Haihong, E., LE, Guan, et al. Survey on NoSQL database. Pervasive computing and applications (ICPCA), 2011 6th international conference on. IEEE, 2011. p. 363-366.

[14] Hutchinson, J., Rouncefield, M., & Whittle, J. (2011, May). Model-driven engineering practices in industry. In Proceedings of the 33rd International Conference on Software Engineering (pp. 633-642). ACM.

[15] Izquierdo, J. L. C., & Cabot, J. (2016). JSONDiscoverer: Visualizing the schema lurking behind JSON documents. Knowledge-Based Systems, 103, 52-55.

[16] JSON Generator (2018) http://www.json-generator.com/. Online; 5 July 2018.

[17] Klettke, M., U. Störl, et S. Scherzinger (2015). Schema extraction and structural outlier detection for json-based nosql data stores. Datenbanksysteme für Business, Technologie und Web (BTW 2015).

[18] Maity, B., Acharya, A., Goto, T., & Sen, S. (2018, June). A Framework to Convert NoSQL to Relational Model. In Proceedings of the 6th ACM/ACIS International Conference on Applied Computing and Information Technology (pp. 1-6). ACM.

[19] MongoDB (2018). Mongodb atlas database as a service. https://www.mongodb.com/. Online; 5 November 2019.

[20] Sevilla, Diego Ruiz, Severino Feliciano Morales, and Jesús García Molina. "Inferring versioned schemas from NoSQL databases and its applications." International Conference on Conceptual Modeling. Springer, Cham, 2015.

[21] Chillón, A. H., Ruiz, D. S., Molina, J. G., & Morales, S. F. (2019). A Model-Driven Approach to Generate Schemas for Object-Document Mappers. IEEE Access, 7, 59126-59142.

[22] Object Management Group (2019) https://www.omg.org/ Online; 5 July 2019.