MDA-based approach for NoSQL Databases modelling

Fatma ABDELHEDI[1,2], Amal AIT BRAHIM[1], Faten ATIGUI[3] and Gilles ZURFLUH[1]

[1]Toulouse Institute of Computer Science Research (IRIT), Toulouse Capitole University, Toulouse, France
[2]CBI2 – TRIMANE, Paris, France
[3]CEDRIC-CNAM, Paris, France
{fatma.abdelhedi, amal.ait-brahim, gilles.zurfluh }@irit.fr, faten.atigui@cnam.fr

Abstract. It is widely accepted today that relational systems are not appropriate to handle Big Data. This has led to a new category of databases commonly known as NoSQL databases that were created in response to the needs for better scalability, higher flexibility and faster data access. These systems have proven their efficiency to store and query Big Data. Unfortunately, only few works have presented approaches to implement conceptual models describing Big Data in NoSQL systems. This paper proposes an automatic MDA-based approach that provides a set of transformations, formalized with the QVT language, to translate UML conceptual models into NoSQL models. In our approach, we build an intermediate logical model compatible with column, document and graph oriented systems. The advantage of using a unified logical model is that this model remains stable, even though the NoSQL system evolves over time which simplifies the transformation process and saves developers efforts and time.

Keywords: UML, NoSQL, Big Data, MDA, QVT, Models Transformation.

1 Introduction

Big Data is one of the current and future research themes. Recently, the advisory and research firm Gartner Group outlined the top 10 technology trends that will be strategic for most organizations over the next five years, and unsurprisingly Big Data is mentioned in the list [13]. Relational systems that had been for decades the one solution for all databases needs prove to be inadequate for all applications, especially those involving Big Data [3]. Consequently, new type of DBMS, commonly known as "NoSQL" [2], has appeared. These systems are well suited for managing large volume of data; they keep good performance when scaling up [1]. NoSQL covers a wide variety of different systems that can be classified into four basic types: key-value, column-oriented, document-oriented and graph-oriented. In this paper, we focus on the last three. The first one (key-value) is implicitly considered since all of the mentioned systems extend the concepts of key-value [10].

To motivate and illustrate our work, we present a case study in the healthcare filed. 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).

The lack of a model when creating a database is a key feature in NoSQL systems. In a table, attributes names and types are specified as and when the row is entered. Unlike relational systems - where the model must be defined when creating the table - the schema less appears in NoSQL systems. This property offers undeniable flexibility that facilitates the evolution of models in NoSQL systems. But this property concerns exclusively the physical level (implementation) of a database [14]. In information system, the model serves as a document of exchange between end-users and developers. It also serves as a documentation and reference for development and system evolution due to the business needs and typically deployment technologies evolution. Furthermore, the conceptual model provides a semantic knowledge element close to human logic, which guarantees efficient data management [3].

UML is widely accepted as a standard modelling language for describing complex data [3]. In the medical application, briefly presented above, the database contains structured data, data of various types and formats (explanatory texts, medical records, x-rays, etc.), and big tables (records of variables produced by sensors). Therefore, we choose the UML class diagram to design describe the medical data.

The rest of the paper is structured as follows: Section 2 defines our research problem and reviews previous work on models transformation; Section 3 introduces our MDA-based approach; two transformations processes are presented in this section, the first one creates a logical model starting from a UML class diagram, and the second one generates NoSQL physical models from this logical model; Section 4 details our experiments; and Section 5 concludes the paper and announces future work.

2 Research Problem and Related Work

Big Data applications developers have to deal with the question: how to store Big Data in NoSQL systems? To address this problem, existing solutions propose to model Big Data, and then define mapping rules towards the physical level.

In the specific context of a data warehouse, both [9] and [15] have proposed to transform a multidimensional model into a NoSQL model. In [9] the authors defined a set of rules to map a star schema into two NoSQL models: column-oriented and document-oriented. The links between facts and dimensions have been converted using imbrications. Authors in [15] proposed three approaches to map a multidimensional model into a logical model adapted to column-oriented NoSQL systems.

Other studies [5] and [6] have investigated the process of transforming relational databases into a NoSQL model. Li [5] have proposed an approach for transforming a relational database into HBase (column-oriented system). Vajk et al. [6] defined a mapping from a relational model to document-oriented model using MongoDB.

To the best of our knowledge, only few works have presented approaches to implement UML conceptual model into NoSQL systems. Li et al. [11] propose a MDA-based process to transform UML class diagram into column-oriented model specific to HBase. Starting from the UML class diagram and HBase metamodels, authors have proposed mapping rules between the conceptual level and the physical one. Obviously, these rules are applicable to HBase, only. Gwendal et al. [7] describe the mapping between a UML conceptual model and graph databases via an intermediate graph metamodel. In this work, the transformation rules are specific to graph databases used as a framework for managing complex data with many connections. Generally, this kind of NoSQL systems is used in social networks where data are highly connected.

Regarding the state of the art, some of the existing works [5] and [6] focus on relational model that, unlike UML class diagram, lacks of semantic richness, especially through the several types of relationships that exist between classes. Other solutions, [9] and [15] have the advantage to start from the conceptual level. But, the proposed models are Domain-Specific (Data Warehouses system), so they consider fact, dimension, and typically one type of links only. [11] and [7] consider, each, a single type of NoSQL systems (column-oriented in [11] and graph-oriented in [7]). However, it makes more sense to choose the target system according to the user's needs. For example, if processing operations requires access to hierarchically structured data, the document-oriented system proves to be the most adapted solution.

The main purpose of our work is to assist developers in storing Big Data in NoSQL systems. For this, we propose a new MDA-based approach that transforms a conceptual model describing Big Data into several NoSQL physical models. This automatic process allows the developer to choose the system type (column, document or graph) that suits the best with business rules and technical constraints.

3 UMLtoNoSQL APPROACH

Our purpose is to define, to formalize and to automate the storage of Big Data by means of NoSQL systems. For this, we propose UMLtoNoSQL approach that automatically transforms a UML conceptual model into a NoSQL physical model. We introduce a logical level between conceptual (business description) and physical (technical description) levels in which a generic logical model is developed. This logical model exhibits a sufficient degree of platform-independency making possibleits mapping to one or more NoSQL platforms. This model have two main advantages: (1) it describes data according to the common features of NoSQL models, (2) it is independent of technical details of NoSQL systems, this means that the logical level remains stable, even though the NoSQL system evolves over time. In this case, it would be enough to evolve the physical model, and of course adapt the transformation rules; this simplifies the transformation process and saves time for developers.

To formalize and automate UMLtoNoSQL process, we use the Model Driven Architecture (MDA). One of the main aims of MDA is to separate the functional specification of a system from the details of its implementation in a specific platform [4]. This architecture defines a hierarchy of models from three points of view: Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM) [8]. Among these models, we use the **PIM** to describe data hiding all aspects related to the implementation platforms, and the **PSM** to represent data using a specific technical platform.

In our scenario, the UML class diagram and the generic logical model belong to the PIM level. UMLtoNoSQL process transforms the UML class diagram (conceptual PIM) into a generic logical model (logical PIM). At the PSM level, we consider three different physical models that correspond to Cassandra (column-oriented system), MongoDB (document-oriented system) and Neo4j (graph-oriented system). Figure 1 shows the different component of UMLtoNoSQL process.

UMLtoGenericModel (1) is the first transformation in UMLtoNoSQL process. It transforms the input UML class diagram into the generic logical model (2). This model is conform to the generic logical metamodel presented in Section 3.1. GenericModeltoPhysicalModel (3) is the second transformation (Section 3.2) that generates the NoSQL physical models (PSMs) (4) starting from the generic logical model.

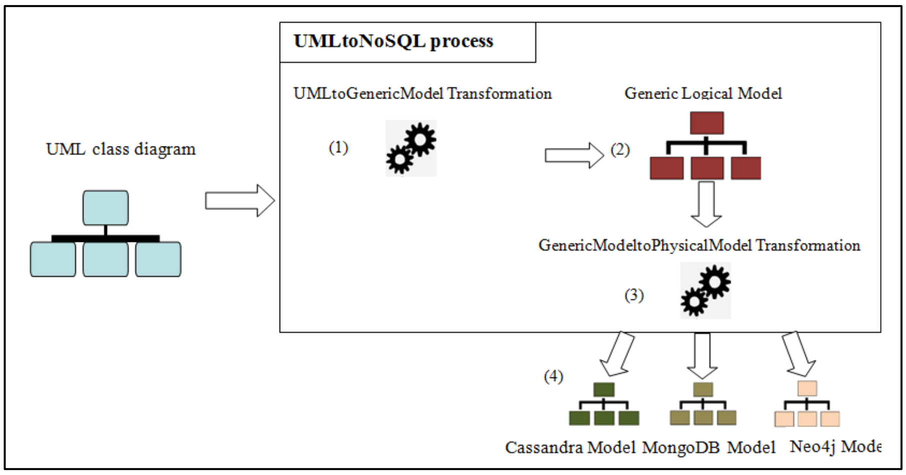


Fig. 1. Overview of UMLtoNoSQL process

3.1 UMLtoGenericModel Transformation

In this section we present the UMLtoGenericModel transformation, which is the first step in our approach as shown in Figure 1. We first define the source (UML Class Diagram) and the target (Generic Logical Model). After that, we focus on the transformation itself.

Source. A Class Diagram (CD) is defined as a tuple (N, C, L), where:
N is the class diagram name,
C is a set of classes. Classes are composed from structural and behavioral features; in this paper, we consider the structural features only. Since the operations describe the behavior, we do not consider them. For each class c ∈ C, the schema is a tuple (N, A, IdentO^c), where:

- c.N is the class name,
- c.A = $\{a_1^c, \dots, a_q^c\}$ is a set of q attributes. For each attribute a^c ∈ A, the schema is a pair (N,C) where "a^c.N" is the attribute name and "a^c.C" the attribute type; C can be a predefined class, i.e. a standard data type (String, Integer, Date ...) or a business class (class defined by user),
- c. IdentO^c is a special attribute of c; it has a name IdentO^c.N and a type called "Oid". In this paper, an attribute which type is "Oid" represents a unique object identifier, i.e. an attribute which value distinguishes an object from all other objects of the same class,

L is a set of links. Each link l between n classes, with n>=2, is defined as a tuple (N, Ty, Pr^l), where:
- l.N is the link name.
- l.Ty is the link type. In this paper, we consider the three main types of links between classes: Association, Composition and Generalization.
- l.Pr^l = $\{pr_1^l, \dots, pr_n^l\}$ is a set of n pairs. ∀ i ∈ {1,..,n}, pr_i^l = (c, cr^c), where pr_i^l.c is a linked class and $pr_i^l.cr^c$ is the multiplicity placed next to c. Note that $pr_i^l.cr^c$ can contain a null value if no multiplicity is indicated next to c (like in generalization link).

Class diagram metamodel is shown in Figure 2; this metamodel is adapted from the one proposed by the OMG [12].

Target. The target of UMLtoGenericModel transformation corresponds to a generic logical model that describes data according to the common features of the three types of NoSQL systems: column-oriented, document-oriented and graph-oriented. In the generic logical model, a DataBase (DB) is defined as a tuple (N, T, R), where:
N is the database name,
T is a set of tables. The schema of each table t ∈ T is a tuple (N, A, IdentL^t), where:

- t.N is the table name,

- t.A = $\{a_1^t, \dots, a_q^t\}$ is a set of q attributes that will be used to define rows of t; each row can have a variable number of attributes. The schema of each attribute $a^t \in$ A is a pair (N,Ty) where "a^t.N" is the attribute name and "a^t.Ty" the attribute type.
- t. IdentL^t is a special attribute of t; it has a name IdentL^t.N and a type called "Rid". In this paper, an attribute which type is "Rid" represents a unique row identifier, i.e. an attribute which value distinguishes a row from all other rows of the same table,

R is a set of binary relationships. In the generic logical model there are only binary relationships between tables. Each relationship r ∈ R between t_1 and t_2 is defined as a tuple (N, Pr^r), where:

- r.N is the relationship name.
- r.Pr^r = $\{pr_1^r, pr_2^r\}$ is a set of two pairs. ∀ i ∈ {1,2}, pr_i^r = (t, cr^t), where pr_i^r.t is a related table and pr_i^r. cr^t is the multiplicity placed next to t.

Metamodel of the proposed generic logical model is shown in Figure 3. Note that the attribute value may be either atomic or complex (set of attributes). We represent this by using the UML XOR constraint.

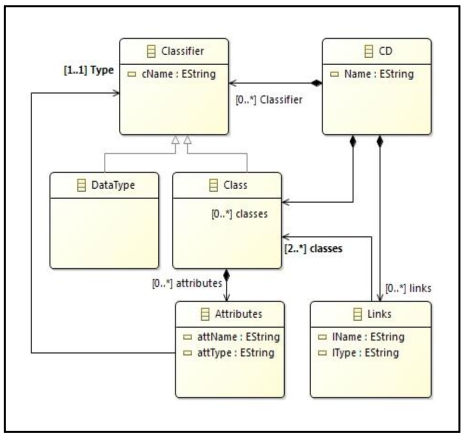


Fig. 2. Source Metamodel

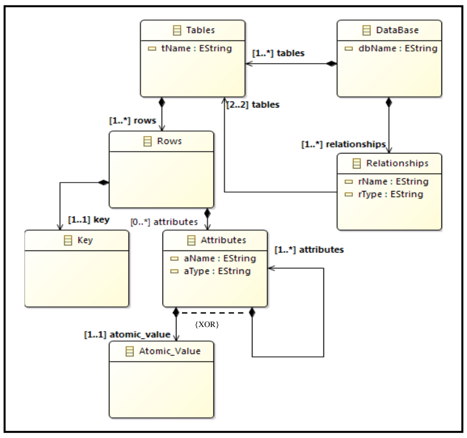


Fig. 3. Target Metamodel

Transformation Rules.

R1: each class diagram CD is transformed into a database DB, where DB.N = CD.N.

R2: each class c ∈ C is transformed into a table t ∈ DB, where t.N = c.N, IdentL^t.N = IdentO^c.N.

R3: each attribute a^c ∈ c.A is transformed into an attribute a^t, where a^t.N = a^c.N, a^t.Ty = a^c.C, and added to the attribute list of its transformed container t such as a^t ∈ t.A.

R4: each binary link l ∈ L (regardless of its type: Association, Composition or Generalization) between two classes c_1 and c_2 is transformed into a relationship r ∈ R between the tables t_1 and t_2 representing c_1 and c_2, where r.N = l.N, r.Pr^r = $\{(t_1, cr^{c_1}),(t_2, cr^{c_2})\}$, cr^{c_1} and cr^{c_2} are the multiplicity placed respectively next to c_1 and c_2.

R5: each link $l \in L$ between n classes $\{c_1, \dots, c_n\}$ (n>=3) is transformed into (1) a new table t^l, where $t^l.N = l.N$ and $t^l.A = \emptyset$, and (2) n relationships $\{r_1, \dots, r_n\}$, $\forall$ i $\in$ {1,..,n} r_i links t^l to another table t_i representing a related class c_i, where $r_i.N$ = (t^l.N)_(t_i.N) and $r_i.Pr^r = \{(t^l, null), (t_i, null)\}$.
R6: each association class c_{asso} between n classes $\{c_1, \dots, c_n\}$ (n>=2) is transformed like a link between multiple classes (R5) using (1) a new table t^{ac}, where $t^{ac}.N = l.N$, and (2) n relationships $\{r_1, \dots, r_n\}$, $\forall$ i $\in$ {1,..,n} r_i links t^{ac} to another table t_i representing a related class c_i, where $r_i.N$ = (t^{ac}.N)_(t_i.N) and $r_i.Pr^r = \{(t^{ac}, null), (t_i, null)\}$. Like any other table, t^{ac} contain also a set of attributes A, where $t^{ac}.A = c_{asso}.A$.

We have formalized these transformation rules using the QVT (Query / View / Transformation), which is the OMG standard for models transformation. An excerpt from QVT rules is shown in Figure 7.

3.2 GenericModeltoPhysicalModel Transformation

In this section we present the GenericModeltoPhysicalModel transformation, which is the second step in our approach UMLtoNoSQL (Figure 1). This transformation creates NoSQL physical models starting from the proposed generic logical model.

Source. The source of GenericModeltoPhysicalModel transformation is the target of the previous UMLtoGenericModel transformation.

Target. To illustrate our approach, we have chosen three well known NoSQL systems: Cassandra, MongoDB and Neo4j.

Cassandra physical model.
In Cassandra physical model, KeySpace (KS) is the top-level container that owns all the elements. It's defined as a tuple (N, F), where:
N is the keyspace name,
F is a set of columns-families. The schema of each columns family $f \in F$ is defined as a tuple (N, Cl, $PrimaryKey^f$), where:

- f.N is the columns-family name,
- f.Cl = $\{cl_1, \dots, cl_q\}$ is a set of q columns that will be used to define rows of f; each row can have a variable number of columns. The schema of each column $cl \in Cl$ is a pair (N,Ty) where "cl.N" is the column name and "cl.Ty" the column type.
- f. $PrimaryKey^f$ is a special column of f; it has a name $PrimaryKey^f$.N and a type $PrimaryKey^f$.Ty (standard data type). $PrimaryKey^f$ identifies each row of f.

MongoDB physical model.
In MongoDB physical model, DataBase (DB^{MD}) is the top-level container that owns all the elements. It's defined as a tuple (N, Cll), where:
N is the database name,

Cll is a set of collections. The schema of each collection cll ∈ Cll is a tuple (N, Fl,Id^{cll}), where:

- cll.N is the collection name,
- cll.Fl = $Fl^A \cup Fl^{CX}$ is a set of atomic and complex fields that will be used to define rows, called documents, of Cll. Each document can have a variable number of fields. The schema of an atomic field $fl^a \in Fl^A$ is a tuple (N,Ty) where "fl^a.N" is the field name and "fl^a.Ty" is the field type. The schema of a complex field $fl^{cx} \in Fl^{CX}$ is also a tuple (N, Fl') where fl^{cx}.N is the field name and fl^{cx}.Fl' is a set of fields where Fl'$\subset$ Fl.
- cll. Id^{cll} is a special field of cll; it has a name Id^{cll}.N and a type Id^{cll}.Ty (standard data type). Id^{cll} identifies each document of cll.

Neo4j physical model.
In Neo4j physical model, Graph (GR) is the top-level container that owns all the elements. It's defined as a tuple (V, E), where:
V is a set of vertex. The schema of each vertex v ∈ V is a tuple (L, Pro, Id^v), where:

- v.L is the vertex label,
- v.Pro = $\{pro_1, \ldots, pro_q\}$ is a set of q properties. The schema of each property pro ∈ Pro is a pair (N,Ty), where "pro.N" is the property name and "pro.Ty" the property type.
- v. Id^v is a special property of v; it has a name Id^v.N, a type Id^v.Ty and the constraint "Is Unique ". It identifies uniquely v in the graph.

E is a set of edges. The schema of each edge e ∈ E is a tuple (L, v_1, v_2), where:
- e.L is the edge label,
- e. v_1 and e. v_2 are the vertexes related by e.

Transformation Rules.
For some NoSQL systems, many solutions can ensure the implementation of the generic logical model. In order to choose the most suitable solution, the developer can be well guided thanks to the performance measurement shown in Section 4.2. These measurements concern the response time of queries that access two related tables; the relationship between these tables has being implemented according to the different solutions shown below. The developer will make his choice according to the queries features he needs to perform as well as the expected performances.

We note that the set of solutions proposed in this section is not inclusive; more marginal solutions may be considered.

To Cassandra physical model.
R1: each database DB is transformed into a keyspace KS, where KS.N = DB.N.
R2: each table t ∈ DB is transformed into a columns-family f ∈ KS, where f.N = t.N, PrimaryKey^f.N = IdentL^t.N.
R3: each attribute a^t ∈ t.A is transformed into a column cl, where cl.N = a^t.N, cl.Ty = a^t.Ty, and added to the column list of its transformed container f such as cl ∈ f.Cl.

R4: As Cassandra does not support imbrication; the only solution we can use to express relationships between columns-families consists in using reference columns. A reference column is a monovalued or multivalued column in one columns-family whose values must have matching values in the primary key of another columns-family; we note that this constraint is not automatically managed by the system Cassandra; it remains the responsibility of the user to check it.

For each relationship r between two tables t_1 and t_2, three solutions could be considered:

Solution 1: r is transformed into a reference column cl referencing f_2 (the columns-family representing t_2), where cl.N = (f_2.N)_Ref and cl.Ty = PrimaryKey^{f2}.Ty, and then added to the columns list of f_1 (the columns-family representing t_1) such as cl ∈ f_1.Cl. While instantiating f_1, the value of the reference column cl will correspond to one or many values in the primary key of f_2.

Solution 2: r is transformed into a reference column cl referencing f_1 (the columns-family representing t_1), where cl.N = (f_1.N)_Ref et cl.Ty = PrimaryKey^{f1}.Ty, and then added to the columns list of f_2 (the columns-family representing t_2) such as cl ∈ f_2.Cl. While instantiating f_2, the value of the reference column cl will correspond to one or many values in the primary key of f_1.

Solution 3: r is transformed into a new columns-family f composed of two reference columns referencing the columns-families f_1 and f_2 representing the related tables t_1 and t_2, where f.N = r.N, f.Cl = $\{cl_1, cl_2\}$, cl_1.N = (f_1.N)_Ref, cl_1.Ty = PrimaryKey^{f1}.Ty, cl_2.N = (f_2.N)_Ref and cl_2.Ty = PrimaryKey^{f2}.Ty.

A reference column can either be monovalued or multivalued. Table 1 indicates the type of the reference column according to the relationship cardinalities and the transformation solution used.

Relationship	Solution	Reference column type
r = (N,{(t_1,*),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Multivalued
	Solution 3	Monovalued
r = (N,{(t_1,1),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Monovalued
	Solution 3	Monovalued
r = (N,{(t_1,*),(t_2,*)})	Solution 1	Multivalued
	Solution 2	Multivalued
	Solution 3	Monovalued

Table 1. Descriptive table of reference column types

To MongoDB physical model.

R1: each database DB is transformed into a MongoDB database DB^{MD}, where DB^{MD}.N = DB.N.

R2: each table t ∈ DB is transformed into a collection cll ∈ DB^{MD}, where cll.N = t.N et Id^{cll}.N = IdentL^{t}.N.

R3: each attribute $a^t \in$ t.A is transformed into an atomic field fl^a, where fl^a.N = a^t.N, fl^a.Ty = a^t.Ty, and added to the field list of its transformed container cll such as fl ∈ cll. Fl^A.
R4: relationships in MongoDB could be transformed by using reference fields or embedding. A reference field is a monovalued or multivalued field in one collection whose values must have matching values in the Id of another collection; checking this constraint remains the responsibility of the user.

For each relationship r between two tables t_1 and t_2, five solutions could be considered:
Solution 1: r is transformed into a reference field fl referencing cll_2 (the collection representing t_2), where fl.N = (cll_2.N)_Ref and fl.Ty = Id^{cll_2}.Ty, and then added to the fields list of cll_1 (the collection representing t_1) such as fl ∈ $cll_1.Fl^A$.
Solution 2: r is transformed into a reference field fl referencing cll_1 (the collection representing t_1), where fl.N = (cll_1.N)_Ref and fl.Ty = Id^{cll_1}.Ty, and added to the field list of cll_2 (the collection representing t_2) such as fl ∈ $cll_2.Fl^A$.
Solution 3: r is transformed by embedding the collection cll_2 representing t_2 in the collection cll_1 representing t_1, where $cll_2 \in cll_1.Fl^{CX}$.
Solution 4: r is transformed by embedding the collection cll_1 representing t_1 in the collection cll_2 representing t_2, where $cll_1 \in cll_2.Fl^{CX}$.
Solution 5: r is transformed into a new collection cll, where cll.N = r.N, cll.Fl = $\{fl_1, fl_2\}$, fl_1.N = (cll_1.N)_Ref, fl_1.Ty = Id^{cll_2}.Ty, fl_2.N = (cll_2.N)_Ref and fl_2.Ty = Id^{cll_2}.Ty, where cll_1 and cll_2 are the collections representing t_1 and t_2.

Each reference field used in Solution 1, 2 and 5 can either be monovalued or multivalued. Table 2 indicates the type of the reference field according to the relationship cardinalities and the transformation solution used.

Relationship	Solution	Reference field type
r = (N,{(t_1,*),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Multivalued
	Solution 5	Monovalued
r = (N,{(t_1,1),(t_2,1)})	Solution 1	Monovalued
	Solution 2	Monovalued
	Solution 5	Monovalued
r = (N,{(t_1,*),(t_2,*)})	Solution 1	Multivalued
	Solution 2	Multivalued
	Solution 5	Monovalued

Table 2. Descriptive table of reference field types

To Neo4j physical model.
R1: each table t ∈ DB is transformed into a vertex v ∈ V, where v.L = t.N, Id^v.N = $IdentL^t$.N.
R2: each attribute $a^t \in$ t.A is transformed into a property pro, where pro.N = a^t.N, pro.Ty = a^t.Ty, and added to the property list of its transformed container v such as pro ∈ v.Pro.

R3: Each relationship r between two tables t_1 and t_2 is transformed into an edge e, where e.L = r.N, relating two vertex v_1 and v_2, where v_1 and v_2 are the vertex representing t_1 and t_2.

4 Experiments

In this section, we show how to transform a UML conceptual model into NoSQL physical models. As presented in section 3.2, several solutions can ensure this transformation; we therefore began by implementing the UMLtoNoSQL process according to each proposed solution, and then we evaluated their performances to assist the developer in choosing the most effective one.

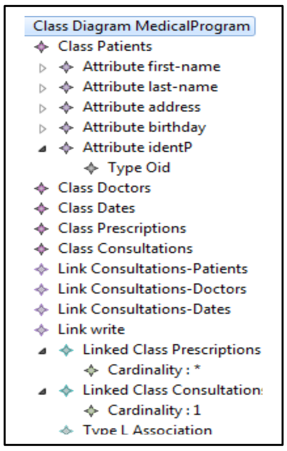
4.1 Implementation

Experimental environment. We carry out the experimental assessment using a model transformation environment called Eclipse Modeling Framework (EMF). It's a set of plugins which can be used to create a model and to generate other output based on this model. Among the tools provided by EMF we use: (1) *Ecore:* the metamodeling language that we used to create our metamodels. (2) *XML Metadata Interchange (XMI):* the XML based standard that we use to create models. (3) *Query / View / Transformation (QVT):* the OMG language for specifying model transformations.

Implementation of UMLtoGenericModel Transformation. UMLtoGenericModel transformation is expressed as a sequence of elementary steps that builds the resulting model (generic logical model) step by step from the source model (UML class diagram):
Step 1: we create Ecore metamodels corresponding to the source (Figure 2) and the target (Figure 3).
Step 2: we build an instance of the source metamodel. For this, we use the standard-based XML Metadata Interchange (XMI) format (Figure 4).
Step 3: we implement the transformation rules by means of the QVT plugin provided within EMF. An excerpt from the QVT script is shown in Figure 7; the comments in the script indicate the rules used.
Step 4: we test the transformation by running the QVT script created in step 3. This script takes as input the source model builded in step 2 and returns as output the logical model. The result is provided in the form of XMI file as shown in Figure 5.

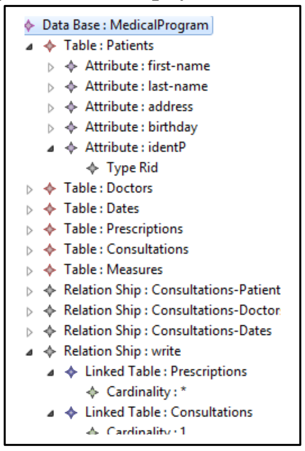
Implementation of GenericModeltoPhysicalModel Transformation. The generic logical model that we proposed in this paper does not imply a specific system; it exhibits a sufficient degree of independence so as to enable its mapping to different NoSQL platforms. For some NoSQL systems, relationships could be transformed into several forms (monovalued or multivalued references, embedding). Lacks of place, we only present one implementation of the generic logical model that was performed on Cassandra according to Solution 1. Figure 8 shows the corresponding QVT script.

This script takes as input the logical model (Figure 5) generated by the previous transformation and return as output Cassandra physical model (Figure 6).



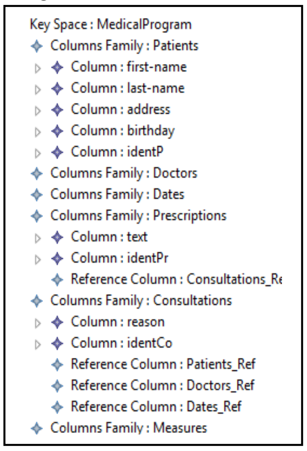
```
Class Diagram MedicalProgram
  Class Patients
    Attribute first-name
    Attribute last-name
    Attribute address
    Attribute birthday
    Attribute identP
      Type Oid
  Class Doctors
  Class Dates
  Class Prescriptions
  Class Consultations
  Link Consultations-Patients
  Link Consultations-Doctors
  Link Consultations-Dates
  Link write
    Linked Class Prescriptions
      Cardinality : *
    Linked Class Consultation
      Cardinality : 1
    Type L Association
```

Fig. 4. Source Model



```
Data Base : MedicalProgram
  Table : Patients
    Attribute : first-name
    Attribute : last-name
    Attribute : address
    Attribute : birthday
    Attribute : identP
      Type Rid
  Table : Doctors
  Table : Dates
  Table : Prescriptions
  Table : Consultations
  Table : Measures
  Relation Ship : Consultations-Patient
  Relation Ship : Consultations-Doctor
  Relation Ship : Consultations-Dates
  Relation Ship : write
    Linked Table : Prescriptions
      Cardinality : *
    Linked Table : Consultations
      Cardinality : 1
```

Fig. 5. Target Model



```
Key Space : MedicalProgram
  Columns Family : Patients
    Column : first-name
    Column : last-name
    Column : address
    Column : birthday
    Column : identP
  Columns Family : Doctors
  Columns Family : Dates
  Columns Family : Prescriptions
    Column : text
    Column : identPr
    Reference Column : Consultations_Re
  Columns Family : Consultations
    Column : reason
    Column : identCo
    Reference Column : Patients_Ref
    Reference Column : Doctors_Ref
    Reference Column : Dates_Ref
  Columns Family : Measures
```

Fig. 6. Cassandra Model

```
modeltype UML uses "http://UMLClassDiagram.com";
modeltype COLM uses "http://GenericLogicalModel.com";
transformation
TransformationUmlToColumnsOrientedModel(in Source:
UML, out Target: COLM);main()
{Source.rootObjects()[ClassDiagram] -> map toDataBase();}
-- Transforming Class Diagram to DataBase
mapping ClassDiagram::toDataBase():DataBase{name :=
self.name;table:=self.classes -> map
toClass();relationship:=self.links -> toRelationship();}
-- Transforming Class to Table
mapping UML
::Class::toClass():COLM::Table{name:=self.name;attributet:=
self.attributec -> map toAttribute();}
-- Transforming Attribute to Column
mapping UML
::Attribute::toAttribute():COLM::Attribute{name:=self.name;
typea:=self.typea -> map toType(); }
mapping UML
::Type::toType():COLM::Type{typea:=self.typea;}
mapping UML
::Link::toRelationship():COLM::RelationShip{name:=self.na
me; linkedtable:=self.linkedclass -> map toLinkedTable();
```

Fig. 7. UMLtoGenericModel

```
TransformationTransformationGenericModelToCassan
draModel(in Source: LogicalPIM, out Target:
CassandraPSM);
main() {
Source.rootObjects()[DataBase] -> map toKeySpace();}
-- Transforming DataBase to KeySpace
mapping DataBase::toKeySpace():KeySpace{
name := self.name;
columnsfamily:=self.table ->
map toColumnsFamily();}
-- Transforming Table to Columns-Family
mapping LogicalPIM
::Table::toColumnsFamily():CassandraPSM::
ColumnsFamily{name:=self.name;column:=self.attribute
t -> map toColumn();referencecolumn:=self.islinkedto ->
map toReferenceColumn();}
mapping LogicalPIM
::Type::toType():CassandraPSM::Type{
if(self.typea =
"Rid"){type:='Int';}endif;type:=self.typea;}
-- Transforming (1,*) RelationShip to a Monovalued Ref
mapping LogicalPIM
::LinkedToTable::toReferenceColumn():CassandraPSM
::ReferenceColumn{if(self.cardinalityoftable = "*" and
self.cardinalityoflinkedtable
```

Fig. 8. GenericModeltoCassandraModel

4.2 Evaluation

The graph-oriented system Neo4j does not offer many solutions to implement relationships; therefore, the developer does not need to choose between several solutions. For Cassandra and MongoDB, where many choices are available, we have evaluated the transformation solutions proposed in section 3.2. This evaluation aims at studying the impact that the choice of the used solution may have on the queries execution time.

Experimental environment. The experiments are done on a cluster made up of 3 machines. Each machine has the following specifications: Intel Core i5, 8GB RAM and 2TB disk.

Data set. In order to perform our experiments, we have used data generator tools. We have generated a dataset of about 1TB with CSV format for Cassandra and JSON format for MongoDB. These files are loaded into the systems using shell commands.

Queries set. For our experiments, we have written 6 queries; each query concerns two tables and the relationship between them. The complexity of these queries increases gradually. The simplest one applies a filter to a table and returns attributes of the other table; the most complex one applies several filters and returns attributes of the two related tables. We note that the concepts "table" and "attribute" correspond respectively to "columns-family" and "column" in Cassandra or "collection" and "field" in MongoDB. An excerpt from our experiment results is depicted in Table 3 and figures 8 (a), (b) and (c). For each query, we indicate the response time obtained according to (1) the relationship cardinalities and (2) the solution used.

NoSQL System	Relationship	Solution	Query	Time(s)
Cassandra	r = (N,{(t_1,*),(t_2,1)})	Solution 1	Q1	140
			Q4	980
			…	
		Solution 2	Q1	830
			…	
		Solution 3	…	
			Q4	420
	r = (N,{(t_1,1),(t_2,1)})	Solution 1	…	
			Q5	310
		Solution 2	…	
			Q6	290
		Solution 3	…	
			Q4	735
	r = (N,{(t_1,*),(t_2,*)})	Solution 1	…	
			Q2	720
			Q5	1400
		Solution 2	…	
			Q3	1050
		Solution 3	…	
			Q5	510
			Q6	530
MongoDB	r = (N,{(t_1,*),(t_2,1)})	Solution 1	…	
			Q4	4300
		Solution 2	…	
			Q6	6200
		Solution 3	Q5	1700
			Q6	1500
		Solution 4	Q2	870
	…			

Table 3. Queries response time

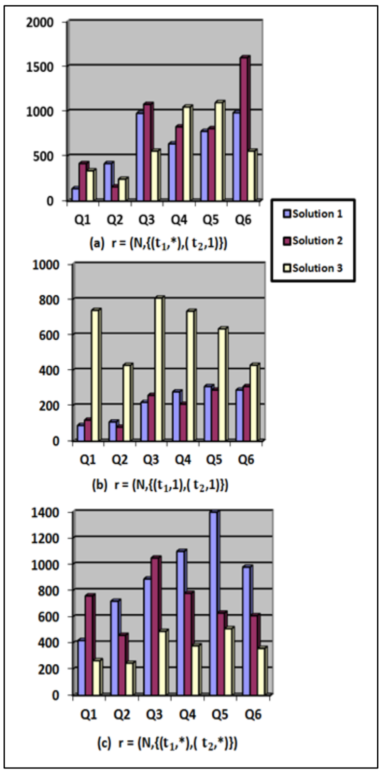


Fig. 9. Cassandra Experimental Results

5 Conclusion and Future Work

In this paper we have proposed an automatic approach that guides and facilitates the Big Database implementation task within NoSQL systems. This approach is based on MDA especially known as a framework for models automatic transformations. Our approach provides a set of transformations that generate a NoSQL physical model starting from a UML conceptual model. In our approach, we build an intermediate logical model compatible with column, document and graph oriented systems; this model uses tables and binary relationships that link them. The independence between the three physical models is ensured. The advantage of using a unified logical model is that this model remains stable, even though the NoSQL system evolves over time. In this case, it would be enough to evolve the physical models, and of course adapt the transformation rules. Furthermore, we have proposed different solutions to transform the binary relationships of the logical model under Cassandra and MongoDB. Depending on the systems functionalities, the binary relationships could be converted into different forms. We have measured the queries response time using each of the proposed solution. The developer can choose the most suited solution according to: (1) Queries features (number of filters, number of attributes to return, etc.), (2) The time response and (3) Query frequency of use.

As future work, we plan to complete our transformation process in order to take into account the constraints of the conceptual level and to preserve the semantics of links when transforming the conceptual model to the logical one. Furthermore, we want to define the transformations rules of physical models into NoSQL scripts using model-to-text transformation (M2T).

6 References

1. A. Angadi, Ak. Angadi, Karuna. Gull. “Growth of New Databases & Analysis of NOSQL Datastores”. In IJARCSSE, 2013.
2. R. Cattell. “Scalable SQL and NoSQL data stores”. ACM SIGMOD Record 3, 2011.
3. A. Abello, “Big Data Design”. In DOLAP. 2015.
4. J. Hutchinson, M. Rouncefield. "Model-driven engineering practices in industry". In ICSE, 2011.
5. C. Li. "Transforming relational database into HBase: A case study". In ICSESS, 2010.
6. T. Vajk, P. Feher, K. Fekete, H. Charaf. “Denormalizing data into schema-free databases”. In CogInfoCom, 2013.
7. D. Gwendal, S. Gerson, C. Jordi. “UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases”. In ER, 2016.
8. J. Bézivin,O. Gerbé. "Towards a Precise Definition of the OMG/MDA Framework". In ASE, 2001.
9. M. Chevalier, M. El Malki, A. Kopliku, O. Teste, R. Tournier. "Implementing Multidimensional Data Warehouses into NoSQL". In ICEIS, 2015.
10. D.Abadi , S.Madden , N.Hachem, "Column-stores vs. row-stores: how different are they really?". In COMAD, 2008.
11. Y. Li, P. Gu, C. Zhang. "Transforming UML Class Diagrams into HBase Based on Meta-model”. In ISEEE, 2014.
12. http://www.omg.org/spec/UML/2.5/
13. http://www.gartner.com/smarterwithgartner/gartners-top-10-technology-trends-2017/
14. V.Herrero, A.Abelló, O.Romero.“NOSQL Design for Analytical Workloads”. In ER 2016.
15. K. Dehdouh, F. Bentayeb, O. Boussaid, N. Kabachi. “Using the column oriented model for implementing big data warehouses”. In PDPTA, 2015.