Journal of Theoretical and Applied Information Technology

15th December 2023. Vol.101. No 23
© 2023 Little Lion Scientific

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

AUTOMATING DATA WAREHOUSE DESIGN WITH MDA APPROACH USING NOSQL AND RELATIONAL SYSTEMS

LAMYA OUKHOUYA[1] , ANASS EL HADDADI[2] ,BRAHIM ER-RAHA[1] , ASMA SBAI[3]

[1]ESTIDMA team, National School of Applied Sciences, Agadir, Morocco

[2]SDIC team, National School of Applied Sciences, Al Hoceima, Morocco

[3]LBH Laboratory, Faculty of Medicine and Pharmacy, Marrakech, Morocco

E-mail: [1]l.oukhouya@uiz.ac.ma, [2]a.alhaddadi@uae.ac.ma, [1]b.erraha@uiz.ac.ma , [3]asma.sbai@uca.ac.ma

ABSTRACT

Data warehouses and OLAP systems provide methods and tools for analyzing data from enterprise information systems. Unfortunately, relational data warehouses are unable to store and analyze data with the 3V characteristics of Big Data: volume, variety, and velocity. To address this, NoSQL systems are introduced in addition to RDBMS, offering scalability to data warehouses to effectively adapt to the volume and variety of collected data. However, integrating these two systems in the same architecture in Big Analytics processes is complex, both in terms of data modeling and data processing. In this regard, several approaches have been proposed to alleviate this complexity. However, several points, which relate to integrated modeling abstractions, adapting the conceptual model with various NoSQL and relational systems, or automating the design process, remain unexamined. In this article, our approach Accounts for all these limitations through a model-driven architecture approach (MDA). This approach proposes a design with three levels of abstraction: conceptual, logical, and physical. The conceptual level is presented by a multidimensional model. The logical level is described by a generic model for all NoSQL and relational families, and the physical level is described by three models related to the implementation; MySQL DBMS for relational systems, and Cassandra DBMS and MongoDB DBMS for NoSQL systems. Moreover, the entire design process is automated through a set of implemented transformation rules rom the conceptual model to source code extraction, thereby facilitating the design task for developers. Furthermore, we conducted a qualitative evaluation compared to other methodologies, revealing that our approach excels in using a generic logical model that can be adapted at the physical level to five types of NoSQL systems and relational systems. Additionally, the automation of the transition from a conceptual model to source code extraction offers a notable advantage in simplifying the migration between concepts.

Keywords: *Data Warehouse, Relational Systems, Nosql Systems, Hybrid Architecture, Model-Driven Architecture.*

1. INTRODUCTION

Decision support systems have evolved in terms of infrastructure and technologies adopted given the growth of data volumes. This evolution has allowed the traditional architecture of decision-making systems to manage Big Data more efficiently and effectively. Data storage is among the elements affected by this evolution. Indeed, information architects have several possible storage modes such as RDBMS, NoSQL, Apache Hadoop [1]. These storages are not exclusively linked and can even coexist with each other. As such, the most striking evolution of decision-making systems is the hybridization of storage modes [2].

The design of a hybrid storage architecture makes it possible to bridge the boundaries of RDBMS, which is the most adopted system, either by data warehouses or in the data sources used by the decision support systems. Since the RDBMS uses a relational model, the new data formats cannot be structured or supported by the model [3]. The implementation of relational models with NoSQL models solves this problem. Indeed, NoSQL Systems have an unstructured storage format; they do not assign a format or schema when storing data. Scalability in storage and processing also characterizes these systems, allowing for great flexibility and performance in data processing [4].

The hybridization of relational systems with NoSQL systems has made it possible to migrate to distributed architectures, offering new approaches to design massive data warehouses in order to effectively adapt to the scalability of the data volume, the variety of data sources, and data velocity. Put differently, hybridization has made it possible to modernize the architecture of decision support systems and to acquire modern 3V data warehouses. In this article, we propose a hybrid storage architecture that integrates both NoSQL and relational systems into the data warehouse design process.

Moreover, designing a data warehousing project presents certain difficulties. These result from the transformation of the multidimensional model, which represents the conceptual level of multidimensional modeling to the other levels, namely the logical and physical levels, which describe the technology chosen for the implementation of the data warehouse. To facilitate this, the MDA approach was proposed as a way to reduce the complexity of data warehouse development [5]. This approach automates transitions between different levels of abstraction, conceptual, logical, and physical, and ultimately leads to the generation of the source code of the design, based on the DBMS chosen for implementation [6]. Furthermore, the similarity between the levels of abstraction of this approach with the three levels of multidimensional modeling offers more efficient management throughout the project life cycle.

This article presents an approach to design a hybrid decision-making architecture on two relational storage systems. Section 2 presents the state of the art. Section 3 describes the approach used to design a relational and NoSQL data warehouse. Section 4 describes the experimentation and validation of our transformation. Section 5 validates our approach by means of a quantitative study. Finally, section 6 presents our conclusion and future work.

2. RELATED WORK

In this section, we will examine the different works involved in the design of NoSQL and relational data warehouses, or the approaches presenting a hybridization with these two technologies.

2.1 Design of relational data warehouse

In article [7], the authors propose a classic approach for designing a data warehouse to apply strategic marketing processes. This approach is based on the use of a snowflake model at the conceptual level, describing the transactional data of the data source. The authors began by presenting the conceptual model used. Then, they described the results of the ETL process ensuring the migration of transactional data to the data warehouse, in accordance with the conceptual model previously designed. In the same context, this work [8] proposes the design of a data warehouse by applying the extraction, transformation, and loading process (ETL) to an operational external data source. In this process, the extraction phase involves reading data from a source file and extracting a required subset from it. The transformation is to convert the extracted/acquired data from its form to the form it must take to be placed in the data warehouse. Finally, the loading phase consists of transferring the processed data to the data warehouse in the form of a multidimensional model with a fact and dimensions. Furthermore, the work [9] proposes a model-driven approach to develop a decision support system, using the canonical multidimensional distribution approach and a conceptual model. This approach is based on three levels. The first level is the CIM, which generates the multidimensional model of the warehouse. This is achieved by applying a set of rules to transform an annotated multidimensional model into a multidimensional conceptual model. The second is the PIM level. It describes the logical level of the implementation, from which the authors chose a ROLAP metamodel for the data warehouse implementation. The transformations at this level correspond to the transition from the multidimensional conceptual diagram produced in the previous level to the multidimensional logical diagram. The last level is the PSM. It represents the implementation of the ROLAP logical model on a relational DBMS, which allows the source code of the data warehouse to be retrieved. All of these transformations are executed with the QVT (Query/ Query/View/Transformation) language dedicated to transformation between models.

A thorough analysis of existing works reveals a general tendency for approaches to be limited to two levels of modelling, either from conceptual to logical, or from logical to physical. While this limitation is practical in a variety of situations, it presents difficulties in ensuring the coherence and flexibility of the entire development process. Adopting an architecture that includes all three levels of abstraction - conceptual, logical and physical, appears to be a crucial solution to overcoming these difficulties. By allowing the separation of concerns between these three levels, this approach aligns perfectly with the Model Driven Architecture (MDA) framework. Isolating conceptual, logical and physical considerations provides greater flexibility, makes it easier for users to use components and develop more coherent and adaptable systems.

Additionally, the majority of current research predominantly concentrates on the ETL model, which is dedicated to data collection, transformation, and loading processes, often overlooking the data structuring model.

Our approach fills this gap by introducing a three-level abstraction architecture that encompasses conceptual, logical and physical aspects. In the context of our study, we place particular emphasis on the design of the physical model of the target database, which is derived from the logical model. This approach enables more efficient data storage and provides a solid basis for the application of advanced analytical processes, making our contribution to the field of database design even more relevant and valuable.

2.2 Design of NoSQL data warehouses

The article [10] presents the design and implementation of a column-oriented data warehouse. The approach used is based on a set of transformation rules making it possible to transform the elements of a multidimensional model into a column-oriented logical model dedicated to the NoSQL Hbase DBMS. These rules transform each fact into a column family and each measure into a unique column belonging to the fact column family. Likewise, dimensions, are converted into column families, and each dimension attribute is converted into a unique column belonging to that column family. The authors also propose operators exploiting the Hadoop MapReduce and Apache Spark paradigms to calculate OLAP cubes. This work presents the design of a massive document-oriented big data warehouse [16]. The approach used by the authors includes two models, successively describing the multidimensional model and the document-oriented model. In addition, a set of transformation rules is proposed, making it possible to convert the multidimensional concepts of the conceptual model into logical concepts dedicated to the document-oriented NoSQL model. In this article [11], the authors examine the creation of an OLAP cube in NoSQL key-value databases. They proposed two approaches; the first is based on the ROLAP technique, which transforms the star schema into a NoSQL logic model in two ways: either the fact and the dimensions are stored in a single key-parent value, or they are stored separately with a kinship. The second approach applies an algorithm inspired by the MOLAP technique to store an OLAP cube under the key-value database based on the bit-coded fragmented storage technique. More recently, this work [12] proposes an approach to transform a multidimensional star model into a NoSQL graph model. To do this, the authors defined a set of de-normalized type transformation rules in order to transform each fact and dimension into a distinct node carrying two labels. The first label mentions the type of component of the conceptual model (fact or dimension), while the second bears the name of the component. Concerning the measurements, they are transformed into properties of the “fact” node. Likewise, the dimension identifier, parameters, and weak attributes are transformed into a property of the "dimension" node. Additionally, the links between dimensions and facts are represented in the NoSQL logic model by a relationship that connects the source node (fact) with the target node (dimension). All these rules are implemented at the physical level with the Talend data integration tool under the NoSQL Neo4j database.

All of these works present the design of a NoSQL data warehouse, specifically, the implementation of a conceptual model on NoSQL systems. However, the data structure used does not offer a logical level, not offering the architecture independence of the result in relation to a particular platform. The key contribution of our approach lies in the design of a logic model that crosses the specificities of each NoSQL family, providing a flexible and universal solution. Unlike previous approaches, which usually focus on models adapted to a specific NoSQL family, our generic logic model provides more portability. As a result, it can be adapted and physically implemented in a variety of contexts, covering all the different data structures typical of each NoSQL family.

2.3 Hybridization of NoSQL and Relational storage modes

The work [13] presents a generic metamodel for relational systems and the 4 NoSQL families implemented at the logical level. the authors present an approach aligned with the MDA approach, where the model transformation is represented by a refinement transformation, applying merge and split regels in order to produce the various possible combinations of the logical model, and this for each family NoSQL and relational systems. In [14], the authors present a generic metamodel for NoSQL and relational systems and modeled with the entity/relationship concept and developed with the Ecore tools of the EMF framework. This metamodel manages aggregations, references, relationship types, and structural variations. In another work [15], the authors propose a functional model with a formal language to integrate graph databases and relational databases. Here, the relationships and their direction in the graph are described as sets of single or multiple values. The process begins in this work by querying the data source with SQL and Cypher languages. Subsequently, the results are adapted and transformed according to the data structures corresponding to the source query.

The research described focuses on hybrid architectures that integrate NoSQL and relational systems seamlessly. These architectures are significantly built around an intermediate level, generally characterised by a metamodel or generic model designed to adapt to both types of system. However, a major problem identified in many existing studies concerns this model, which is often built around the concept of Entity/Relation (E/R), a design choice that is poorly adapted to decision-making contexts. The disadvantages associated with this modelling approach are complexity, performance degradation, difficulties in representing historical data effectively and user comprehension difficulties.

In our approach, these limitations are addressed and consciously resolved. Instead of adopting the E/R model, we use a UML class diagram enriched with multidimensional concepts to represent the generic metamodel. This choice of design offers a representation based on high-level abstractions, providing a level of maturity and flexibility that is crucial for the effective modelling of complex systems. Using a UML class diagram instead of an E/R modelling approach, our methodology not only avoids the drawbacks of previous approaches, but also provides a more flexible and intuitive way of representing complex data relationships in the context of decision making. This strategic change is part of a wider objective to improve the overall efficiency and applicability of hybrid architectures in real-life situations within organizations.

The conclusions drawn from the literature review underline that our approach strategically deals with all the limitations that have been identified in the three categories mentioned above. Our methodology is designed to perform at three distinct levels of abstraction - conceptual, logical and physical, providing a comprehensive and multi-purpose solutions. At the conceptual level, we employ a multidimensional model that aligns perfectly with the data warehousing concepts described with a UML diagram. This allows for a robust representation of high-level abstractions, establishing a solid foundation for the conceptualization of complex data structures. At the logical level, our approach is based on a generic implementation that shows adaptability to the diverse database management system environment. This adaptability integrates the four main families of NoSQL systems as well as traditional relational systems, providing a unified and generic logical framework.
Finally, the physical level consists of a set of database management system (DBMS) models designed specifically to take into account the unique characteristics of NoSQL and relational systems. This complex design enables the physical implementation of conceptual and logical models, optimizing data storage and recovery. Our approach is distinguished above all by its automation capabilities, which are made possible by the MDA (Model-Driven Architecture) approach. This automation provides a transparent transition between levels of abstraction. From conceptual model to source code extraction, each physical model representing the DBMS implementation is systematically generated automatically from the logical model, which is in turn obtained automatically from the conceptual model. This process, described by a series of transformations between levels of abstraction, not only improves efficiency, but also contributes to the general effectiveness of our approach by simplifying the complex task of switching from conceptualization to operational implementation.

In the following section, we will present our hybrid architecture incorporating the various characteristics mentioned earlier.

3. HYBRID DECISION-MAKING ARCHITECTURE WITH NOSQL AND RELATIONAL SYSTEMS

This article seeks to introduce a decision-making architecture for hybrid storage, integrating both relational and non-relational systems. Specifically, our goal is to establish an automated approach for designing a data warehouse compatible with both relational and non-relational systems.

The choice of a hybrid storage architecture is explained by the current need for organizations to navigate through situations where it is imperative to use both systems simultaneously. Let's take the example of an e-commerce company that adopts an approach combining relational and NoSQL database systems in its infrastructure. Relational systems are privileged for managing users and orders, guaranteeing data integrity and financial security thanks to the ACID properties applied by RDBMSs. At the same time, NoSQL systems are playing a vital role in real-time event processing, such as tracking user interactions and integrating them in real time. This merger provides a solution adapted to the specific requirements of each domain, improving the global efficiency and performance of the system.

In designing our hybrid storage architecture, we chose to adopt the MDA approach for several fundamental reasons. This decision was based on the significant advantages offered by the MDA methodology. By allowing a clear definition of the architecture at different levels, this approach ensures unfailing consistency, satisfying our requirement for conceptual clarity. In addition, the flexibility and adaptability inherent in the creation of technology-independent models is essential to cope with the variations in hybrid storage. Finally, interoperability between components and automation between certain phases is a key factor that justifies our preference for the MDA approach, facilitating development while mitigating the overall complexity of implementation.

Furthermore, the application of the MDA approach in our work is represented by the establishment of a set of rules enabling the implementation of a multidimensional model on models relating to the following DBMSs: Cassandra, MongoDB and MySQL. Figure 1 illustrates our prototype.

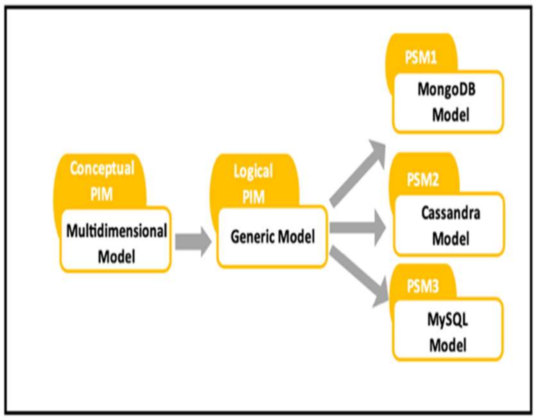


Figure 1: Our Prototype

Our approach is based on the model-driven approach (MDA), which uses 3 levels of abstraction: CIM, PIM, and PSM. In our work, we will only retain the PIM and PSM levels, since our conceptual model is not designed according to a requirement engineering approach,at the PIM level, will use two types of models: a conceptual PIM, and a logical PIM. The first model will reflect the conceptual level, i.e., the multidimensional model of the data warehouse, while the logical model will represent in a generic way relational concepts as well as all NoSQL concepts for the 4 families, namely, column-oriented, document-oriented, key-value-oriented and graph-oriented. Regarding the PSM level, it will describe the 3 models relating to NoSQL and relational platforms. We chose the Cassandra DBMS representing the column-oriented model, the MongoDB DBMS for the document-oriented model, and the MySQL DBMS for the relational model.

Our automated process, named ToCreateDWH, ensures the transition between these different models by involving a set of model-to-model (M2M) type transformations described with the QVT language, and a second type of model-to-text (M2T) transformation.) with the MOF2Text language allowing you to arrive at the source code for designing the data warehouse. Figure 2 illustrates the different components of our process.

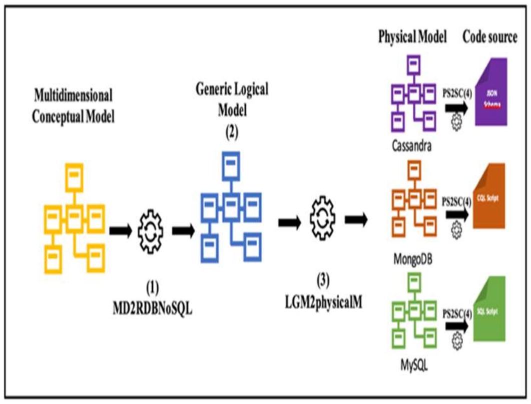


Figure 2: ToCreateDWH Processes Steps

The MD2RDBNoSQL transformation presents the first transformation of our ToCreateDWH process. It makes it possible to transform the multidimensional model into a generic logic model for NoSQL and relational systems. The LGM2physicalM transformation presents the second transformation that generates the physical models relative to the chosen DBMS. Finally, the PS2SC transformation presents the translation of each model to the associated source code. In the next section, will present for each transformation the input, the output, and the associated transformation rules. In the next section, we present the input, output, and associated transformation rules for each

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

transformation. For the PS2SC transformation, which is of the Model-To-Text type, these concepts cannot be presented. Indeed, the MDA approach does not consider this step as a real transformation, which requires an input model and an output model, but this one has only one input, namely the physical model. Therefore, this transformation is described in a code that serves as a translation to the appropriate source code.

3.1 MD2RDBMNoSQL Transformation

In this part, will present the first transformation of the ToCreateDWH process that transforms the conceptual PIM model into a logical PIM model. First, we will describe the source and the target of this transformation, then we will present the associated transformation rules.

3.1.1 Target: Logical PIM

Our conceptual PIM model describes the data structure according to multidimensional concepts. Analysis topics, also called Facts, encompass a set of indicators, known as Measures. The values of these indicators are observed according to what we call the axes of analysis, more commonly called dimensions (Dimension). These dimensions are made up of different levels of granularity, which are organized into hierarchies that we have named base. Each Base is attached to a dimension, which is composed of an identifier (Identifying Attribute), dimension attribute (Dimension Attribute), and optionally, an optional attribute allowing for expressing additional data on the dimension attribute (OptionalAttribute). Figure 3 presents the PIM conceptual model used.

3.1.2 Target: Logical PIM

The objective of our logical model is to create a schema that can be in accordance with various models used at the physical level. The added value of our model is enabled to integrate with all NoSQL families. This means that if a case study is compliant with a particular family, either NoSQL or relational systems, it will be automatically adaptable with the other families thanks to the implementations of our PIM model that are performed at the physical level. Figure 4 presents the constituents of our logical PIM model.This model consists of a concept (Concept) that groups the recordings. The latter can take different forms (depending on the constitutive elements of the physical model of a table, a collection, a record, or a node (in the graph-oriented model).

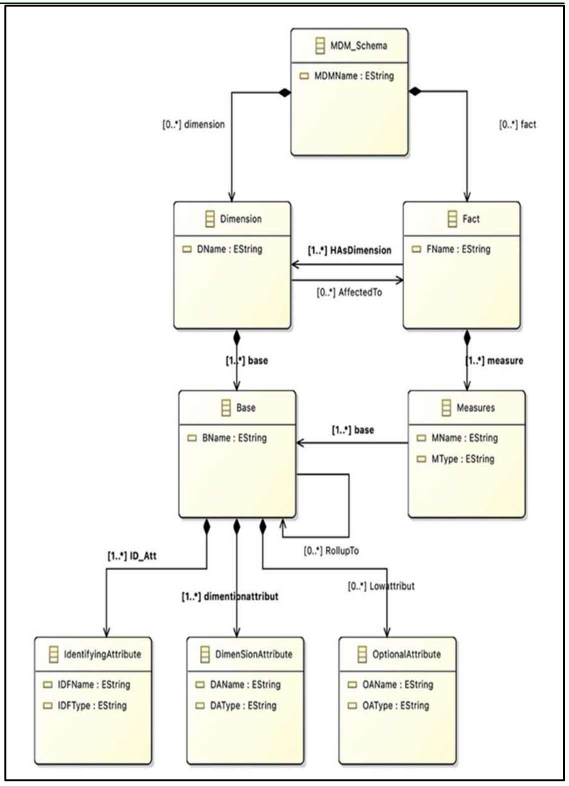


Figure 3 : PIM conceptual metamodel

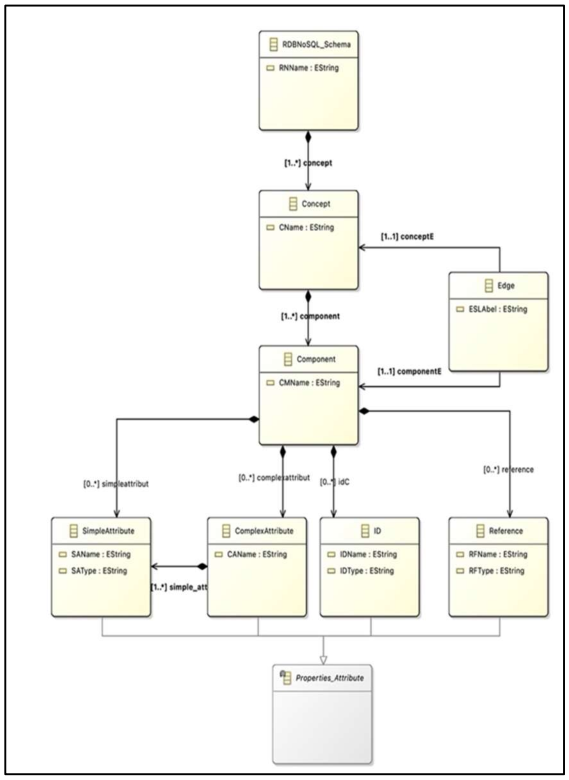


Figure 4 : PIM logical Metamodel

Each concept is defined by several "Component" components that refer to families of columns to documents (in the document-oriented template) or to columns (in the relational template). Attribute properties (Properties Attribute) specify different characteristics for each component, including an identifier (ID) to determine identifier type attributes, an attribute (Simple Attribute) to specify atomic value attributes (such as String, int, float, etc.), and another (Complex Attribute) for nesting attributes, and finally a reference attribute (Reference) that determines the notion of foreign key in the relational model.

3.1.3 Transformation Rules:

After having defined the two source and target models, we will move on to define the transformation rules automating the transition from the conceptual PIM model to the logical PIM model. This passage is realized by a series of transformations described as follows:

R1: Each Multidimensional model is transformed into generic model RDBNoSQl, where name of conceptual model is equivalent to the name of logical model, RDBNoSQL_Schema.CName= MDM_Schema.MDMName.

R2: Each Fact is transformed into a Concept whose Fact name is equivalent to the Concept name,where Fact.FName=Concept.CName.

R3: Each Dimension is transformed into Component of the Logical Model, whose name of the dimension is equivalent to the name of the Component, where Dimension.DName= Component.RName.

Moreover, in order to make the logical model richer in terms of data structuring represented at the physical level (normalized, and denormalized approach), the transformation of the dimension is also duplicated as much as complex attribute, hence the Dimension.DName = ComplexAttributs. CXName.

R4: Each IdentifyingAttribute is transformed into ID, where IdentifyingAttribute.IdName=ID.IDName, and IdentifyingAttribute.IDType=ID.IDType. This element is successively transformed into References, thus representing the dimension identifiers associated with the Fact, thus IdentifyingAttribute.IdName=Reference.RName and IdentifyingAttribute.IDType=Reference.Rtype .

R5: Each DimensionAttributes is transformed into SimpleAttributes, hence DimensionAttributes.DAName=SimpleAttributes.ATName, and DimensionAttributes.DAType = SimpleAttributes.ATType.

R6: Each OptionalAttributes is transformed into SimpleAttributes, hence OptionalAttributes.LAName=SimpleAttributes.ATName, and OptionalAttributes.LAType= SimpleAttributes.ATType.

R7: Each DimensionAttributes is transformed into SimpleAttributes, hence DimensionAttributes.DAName=SimpleAttributes.ATName, and DimensionAttributes.DAType= SimpleAttributes.ATType.

R8: Each Measures is transformed into SimpleAttributes, hence Measures.DName= SimpleAttributes.ATName, and Measures.MType= SimpleAttributes.ATType.

3.2 LGM2physicalM Transformation

This section will present the second transformation relating to the transformation of the PIM logic model representing the source of this transformation to the targets described by the chosen implementation DBMS, namely, Cassandra and NoSQL. We will first introduce the PSM models. Then, we will describe the rules of transition from the generic model to the physical models associated with each of these systems.

3.2.1 Source: logical PIM

The source of this transformation is presented in the previous section (Figure 4) as well as the output for the MD2RDBNoSQL transformation. The input is given the generic PIM logical model with NoSQL and relational systems.

3.2.2 Target: physical PSM

The LGM2physicalM transformation output corresponds to physical NoSQL and relational models. To illustrate our proposal, we chose to implement our generic model on three DBMSs, namely, Cassandra (column-oriented), MongoDB (document-oriented), and MySQL(relational). In this section, we will introduce the data model components for each of these systems.

3.2.2.1 MySQL PSM Metamodel

The MySQL model is designed to represent a relational database. It establishes a predefined schema that must be defined before adding data. Data structuring within this schema is based on several concepts: tables are used to organize rows and columns of data, primary keys are used to uniquely identify values, and foreign keys are used to establish references to values in other tables. Figure 5 presents the MySQL relational model.

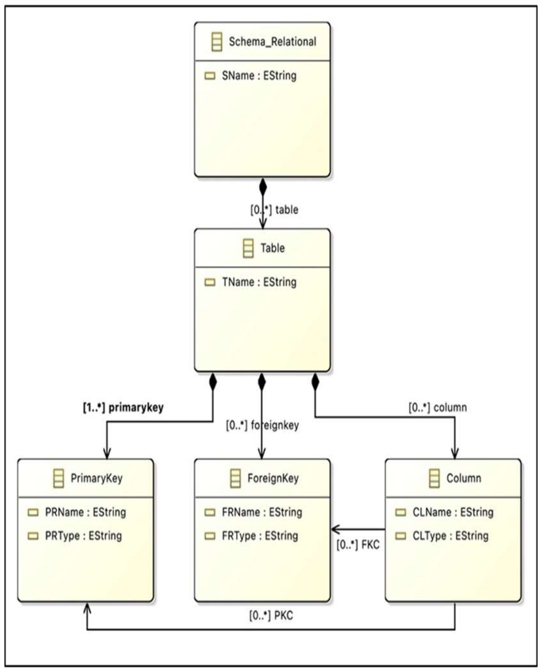


Figure 5: MySQL PSM Metamodel

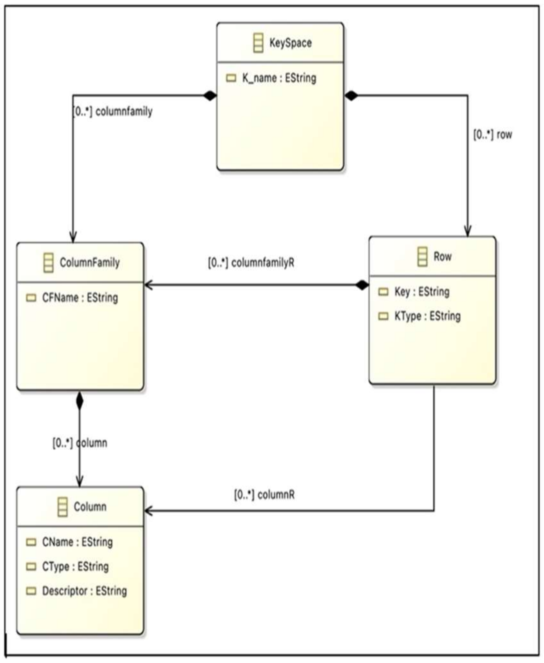


Figure 6: Cassandra PSM Metamodel

3.2.2.2 Cassandra PSM metamodel

The Cassandra model represents a NoSQL database system that follows a column-oriented approach. In this model, the data organization is grouped into a KeySpace that serves as a container for all elements defining the data structure. The main component of this structure is the column family (Column Family), which is defined by both a name and a collection of columns (Column). Each column is characterized by a name, a data type and a descriptor. The descriptor plays a crucial role in identifying the column as a primary key, which is used to uniquely identify rows within a column family. Figure 6 presents a visual representation of the Cassandra metamodel.

3.2.2.3 MongoDB PSM metamodel

MongoDB is a document-driven NoSQL database. In this database, data is organized using a structure consisting of a collection and documents. A collection is defined by a name and contains a set of documents. The structure of a document is specified through fields or attributes, which can be comparable to key/value pairs where the attribute name acts as the key. We can distinguish between atomic fields, whose values are elementary (atomic document), and complex fields, whose values themselves include documents, which are called complex documents. You can find a visual representation of the MongoDB metamodel in Figure 7.

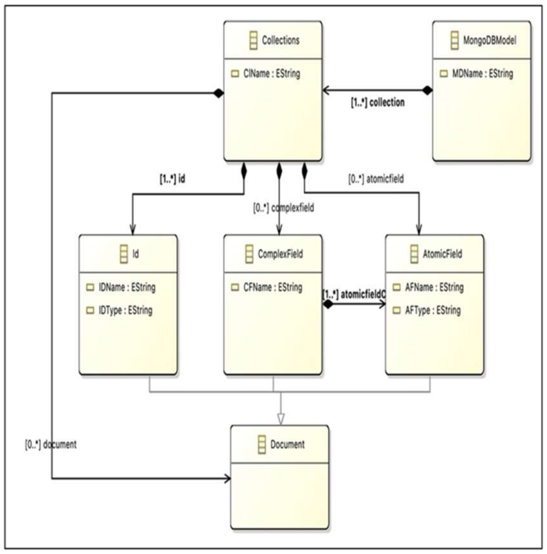


Figure 7 : MongoDB PSM Metamodel

3.2.3 Transformations Rules

3.2.3.1 Transformation To MySQL

The relational model is defined in all RDBMS before the database is populated. That is, the structure of the model is fixed, and the data is inserted as tables, key-primary, foreign-key, and column.

R1: Each model structure is transformed into a MySQL relational data model from which RDBNoSQL_Schema.CName= Schema_Relational. SName.

R2: Each Concept and Component is transformed into a Table, where: Table. Name = Concept.CName and Table.Name= Component.CMName.

R3: Each ID is transformed into a PrimaryKey then added to the appropriate pair, hence PrimaryKey.Name=ID.IDName and PrimaryKey.Type = ID.IDType .

R4: Each Reference is transformed into a Foreignkey, hence ForeignKey.Name =Reference.Name and ForeignKey.Type= Reference.Name .

R5: Each SimpleAttribute is transformed into a Column, hence Column.Name = SimpleAttribute.SAName and Column.Type=SimpleAttribute.SAType.

3.2.3.2 Transformation To Cassandra

Apache Cassandra follows the same philosophy as relational systems, where the data model must be fixed in advance. Thus, the name of the database, the column families and the columns are fixed.

R1: Each Data Schema is transformed into a KeySpace, where: keyspace. Name=Schemadata.Name.

R2: Each Concept and Component is transformed into a distinct Column Family, where: ColumnFamily.Name=Concept.CName and ColumnFamily.Name= Component.CName.

R3: Each Reference is transformed into a Column and added to the corresponding Column Family, where: the Reference.Name=Column.CName, Reference.Type= Column.CType, and Column.Decription= 'primaryKey'.

R4: Each ID is transformed into a Column and added to the corresponding Column family: ID.Name= Column.Name, ID.Type=Column.type and Decription= 'PrimaryKey'. If the ID is represented as the line identifier, and this is in the case of the transformation of the Fact identifier to ID that we saw in the previous section, the transformation will be described as follows: ID.Name=Column.Name, ID.Type= "uuid" and Decription= 'primaryKey'. The “uuid” type is a function used by Cassandra to determine unique line identifiers.

R5: Each Simpleattributes is transformed into a Column and added to the corresponding Column family;hence:Simpleattributes.Name = Column.Name and Simpleattributes.type=column.type.

3.2.3.3 Transformation To MongoDB

Before entering data into the MongoDB database, part of the model is declared. This involves specifying the name of the database and the names of the collections. The other components are translated according to a data normalization approach. It should be specified that the transformations towards the physical level all follow this structuring. The transformation rules are as follows:

R1: Each DataSchema is transformed into a MongoDB schema, hence: MongoDBModel.Name = Schemadata.Name.

R2: Each Concept and Component is transformed into a distinct Collection, hence: Collection.Name=Concept.CName and Collection.Name=Component.CMName

R3: Each ID is transformed into a document identifier Id and added to the corresponding Collections. Thus, Id.name=ID.Name and Id.Ktype= ID.IDType .

R4: Each SimpleAttribute is transformed into an AtomicField, hence: AtomicFeild.Name = Simpleattributes.SAName, and AtomicField. AFvalue = Simpleattributes.SAType.

R5: Each complex attribute is transformed into a ComplexField hence: ComplexField.CFLName = ComplexAttributs.CXName.

After presenting the input and output for each transformation, as well as the transformation regimes, the next section presents the implementation of these transformations.

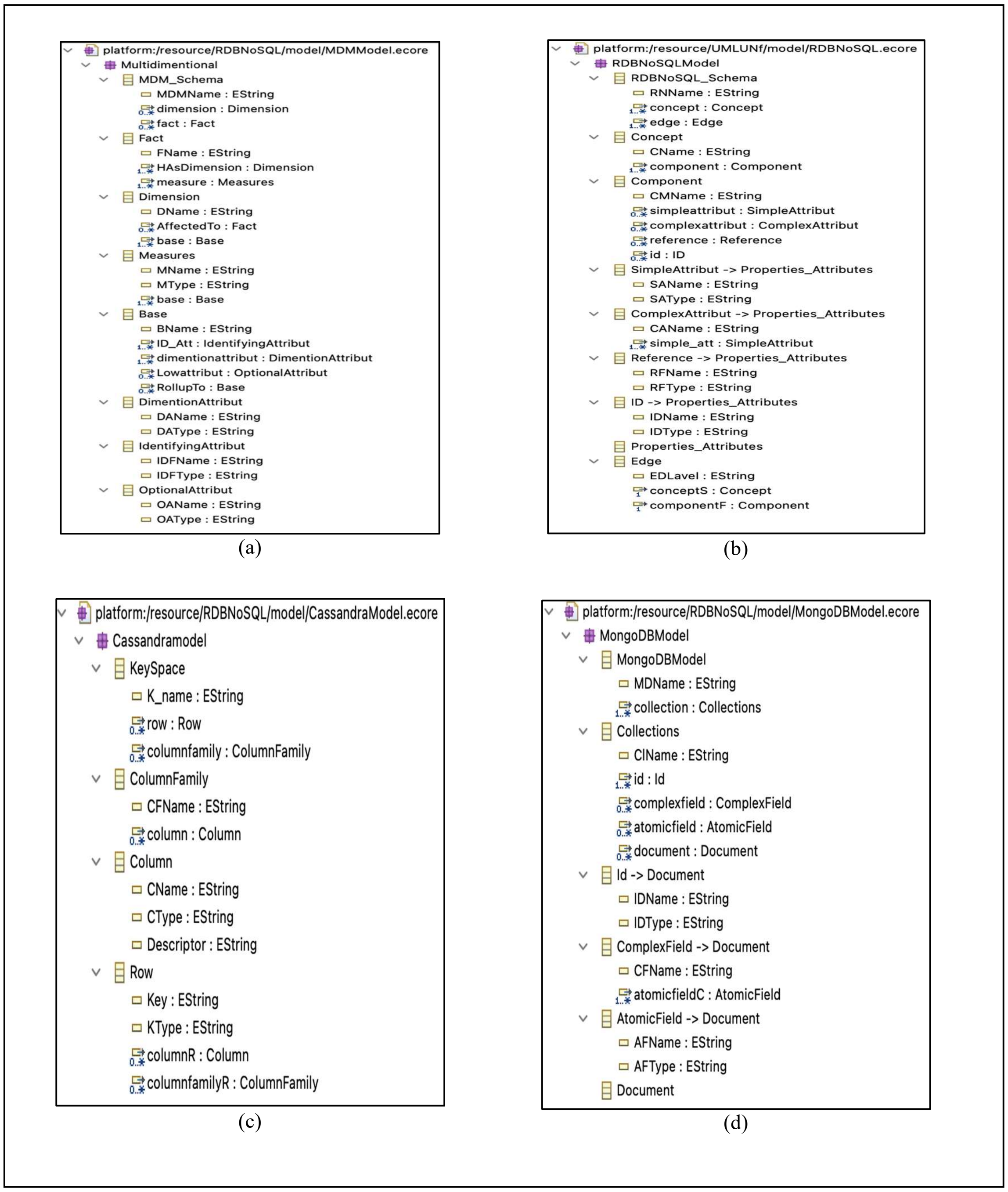
3.3 Implementation of transformation rules

The implementation of the transformations is deployed on the Eclipse platform. Specifically, we used EMF which is a complete environment for the implementation of our MDA approach.Thus, we used the Ecore metamodel to implement our models. Figure 8 show the Ecore models for the conceptual PIM(a), the logical PIM(b), and the three physical PSMs dedicated to DBMS: Cassandra(c), MongoDB(d), and MySQL(e).

Regarding transformations, we chose the QVT language to describe the transformation rules presented in “section A” for the transition from the conceptual model to the logical model, as well as the transformation rules determined in “section B” presenting the transition from the generic model to the physical models related to Cassandra DBMS, MongoDB, and MySQL. Figure 9 shows the QVT code for the first pass and the code for the second pass. Besides, the creation code from each PSM is generated by applying a set of translation rules formalized in MOF2Text code.

This language is based on a model-based approach, where the text generated from the model is defined as a set of configurable text templates with the elements of the model. This indicates for each element of the source model, the elements that correspond to it at the level of the text (code relative to the DBMS of the source model).

Regarding the implementation of this language, we chose the Acceleo project, which is a source code generator to implement the MDA approach to realize applications from models based on EMF. Figure 10 presents an extract of the MOF2Text script for the translation of the Cassandra model into the CQL script, the MongoDB model into the "JSON Schema" script, and finally the MySQL script SQL model.



```
platform:/resource/RDBNoSQL/model/MDMModel.ecore
  Multidimentional
    MDM_Schema
      MDMName : EString
      dimension : Dimension
      fact : Fact
    Fact
      FName : EString
      HAsDimension : Dimension
      measure : Measures
    Dimension
      DName : EString
      AffectedTo : Fact
      base : Base
    Measures
      MName : EString
      MType : EString
      base : Base
    Base
      BName : EString
      ID_Att : IdentifyingAttribut
      dimentionattribut : DimentionAttribut
      Lowattribut : OptionalAttribut
      RollupTo : Base
    DimentionAttribut
      DAName : EString
      DAType : EString
    IdentifyingAttribut
      IDFName : EString
      IDFType : EString
    OptionalAttribut
      OAName : EString
      OAType : EString
```

(a)

```
platform:/resource/UMLUNf/model/RDBNoSQL.ecore
  RDBNoSQLModel
    RDBNoSQL_Schema
      RNName : EString
      concept : Concept
      edge : Edge
    Concept
      CName : EString
      component : Component
    Component
      CMName : EString
      simpleattribut : SimpleAttribut
      complexattribut : ComplexAttribut
      reference : Reference
      id : ID
    SimpleAttribut -> Properties_Attributes
      SAName : EString
      SAType : EString
    ComplexAttribut -> Properties_Attributes
      CAName : EString
      simple_att : SimpleAttribut
    Reference -> Properties_Attributes
      RFName : EString
      RFType : EString
    ID -> Properties_Attributes
      IDName : EString
      IDType : EString
    Properties_Attributes
    Edge
      EDLavel : EString
      conceptS : Concept
      componentF : Component
```

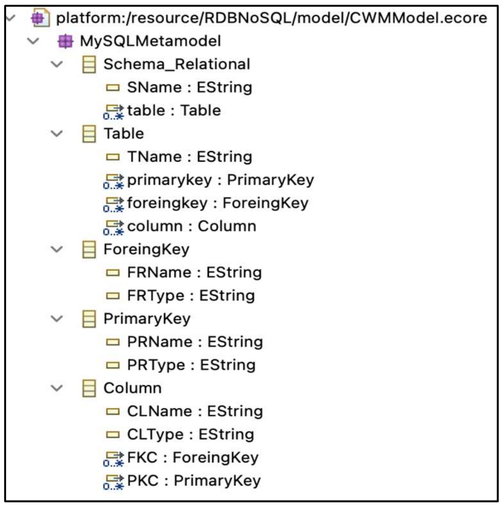
(b)

```
platform:/resource/RDBNoSQL/model/CassandraModel.ecore
  Cassandramodel
    KeySpace
      K_name : EString
      row : Row
      columnfamily : ColumnFamily
    ColumnFamily
      CFName : EString
      column : Column
    Column
      CName : EString
      CType : EString
      Descriptor : EString
    Row
      Key : EString
      KType : EString
      columnR : Column
      columnfamilyR : ColumnFamily
```

(c)

```
platform:/resource/RDBNoSQL/model/MongoDBModel.ecore
  MongoDBModel
    MongoDBModel
      MDName : EString
      collection : Collections
    Collections
      ClName : EString
      id : Id
      complexfield : ComplexField
      atomicfield : AtomicField
      document : Document
    Id -> Document
      IDName : EString
      IDType : EString
    ComplexField -> Document
      CFName : EString
      atomicfieldC : AtomicField
    AtomicField -> Document
      AFName : EString
      AFType : EString
    Document
```

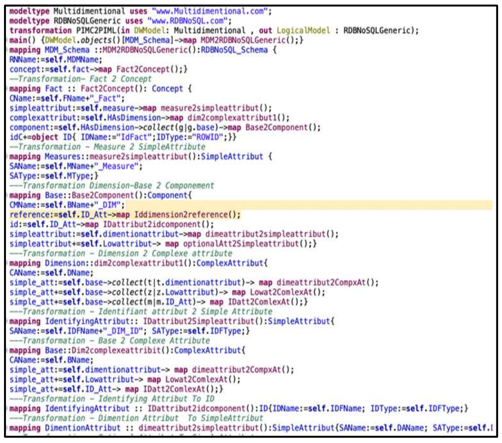
(d)



```
platform:/resource/RDBNoSQL/model/CWMModel.ecore
  MySQLMetamodel
    Schema_Relational
      SName : EString
      table : Table
    Table
      TName : EString
      primarykey : PrimaryKey
      foreingkey : ForeingKey
      column : Column
    ForeingKey
      FRName : EString
      FRType : EString
    PrimaryKey
      PRName : EString
      PRType : EString
    Column
      CLName : EString
      CLType : EString
      FKC : ForeingKey
      PKC : PrimaryKey
```

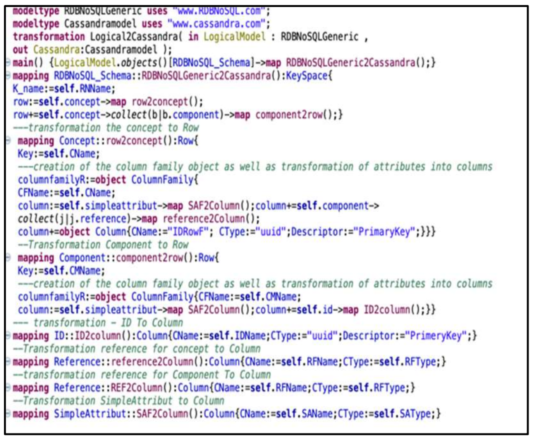
(e)

Figure 8 : ToCreatDWR Ecore Metamodels



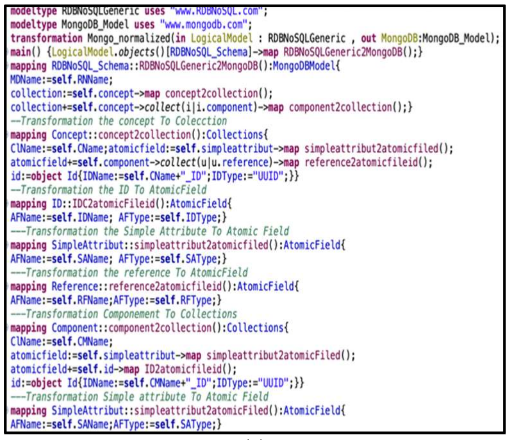
```
modeltype Multidimentional uses "www.Multidimentional.com";
modeltype RDBNoSQLGeneric uses "www.RDBNoSQL.com";
transformation PIMC2PIML(in DWModel: Multidimentional , out LogicalModel : RDBNoSQLGeneric);
main() {DWModel.objects()[MDM_Schema]->map MDM2RDBNoSQLGeneric();}
mapping MDM_Schema ::MDM2RDBNoSQLGeneric():RDBNoSQL_Schema {
RNName:=self.MDMName;
concept:=self.fact->map Fact2Concept();}
--Transformation- Fact 2 Concept
mapping Fact :: Fact2Concept(): Concept {
CName:=self.FName+"_Fact";
simpleattribut:=self.measure->map measure2simpleattribut();
complexattribut:=self.HAsDimension->map dim2complexattribut1();
component:=self.HAsDimension->collect(g|g.base)->map Base2Component();
idC+=object ID{ IDName:="IdFact";IDType:="ROWID";}}
--Transformation - Measure 2 SimpleAttribute
mapping Measures::measure2simpleattribut():SimpleAttribut {
SAName:=self.MName+"_Measure";
SAType:=self.MType;}
---Transformation Dimension-Base 2 Componement
mapping Base::Base2Component():Component{
CMName:=self.BName+"_DIM";
reference:=self.ID_Att->map Iddimension2reference();
id:=self.ID_Att->map IDattribut2idcomponent();
simpleattribut:=self.dimentionattribut->map dimeattribut2simpleattribut();
simpleattribut+=self.Lowattribut-> map optionalAtt2Simpleattribut();}
---Transformation - Dimension 2 Complexe attribute
mapping Dimension::dim2complexattribut1():ComplexAttribut{
CAName:=self.DName;
simple_att:=self.base->collect(t|t.dimentionattribut)-> map dimeattribut2CompxAt();
simple_att+=self.base->collect(z|z.Lowattribut)-> map Lowat2ComlexAt();
simple_att+=self.base->collect(m|m.ID_Att)-> map IDatt2ComlexAt();}
---Transformation - Identifiant attribut 2 Simple Attribute
mapping IdentifyingAttribut:: IDattribut2Simpleattribut():SimpleAttribut{
SAName:=self.IDFName+"_DIM_ID"; SAType:=self.IDFType;}
---Transformation - Base 2 Complexe Attribute
mapping Base::Dim2complexeattribit():ComplexAttribut{
CAName:=self.BName;
simple_att:=self.dimentionattribut-> map dimeattribut2CompxAt();
simple_att+=self.Lowattribut-> map Lowat2ComlexAt();
simple_att+=self.ID_Att-> map IDatt2ComlexAt();}
---Transformation - Identifying Attribut To ID
mapping IdentifyingAttribut :: IDattribut2idcomponent():ID{IDName:=self.IDFName; IDType:=self.IDFType;}
---Transformation - Dimention Attribut  To SimpleAttribut
mapping DimentionAttribut :: dimeattribut2simpleattribut():SimpleAttribut{SAName:=self.DAName; SAType:=self.
```

(a)



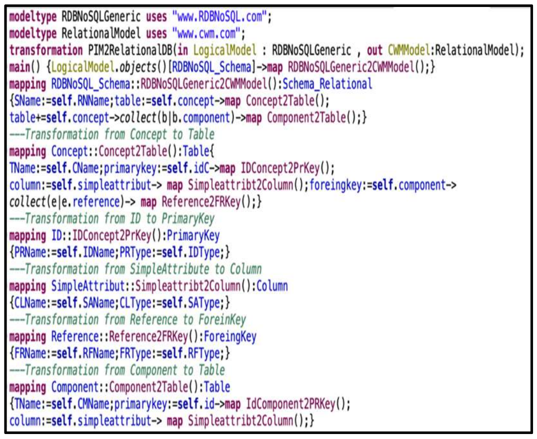
```
modeltype RDBNoSQLGeneric uses "www.RDBNoSQL.com";
modeltype Cassandramodel uses "www.cassandra.com";
transformation Logical2Cassandra( in LogicalModel : RDBNoSQLGeneric ,
out Cassandra:Cassandramodel );
main() {LogicalModel.objects()[RDBNoSQL_Schema]->map RDBNoSQLGeneric2Cassandra();}
mapping RDBNoSQL_Schema::RDBNoSQLGeneric2Cassandra():KeySpace{
K_name:=self.RNName;
row:=self.concept->map row2concept();
row+=self.concept->collect(b|b.component)->map component2row();}
---transformation the concept to Row
 mapping Concept::row2concept():Row{
 Key:=self.CName;
 ---creation of the column family object as well as transformation of attributes into columns
 columnfamilyR:=object ColumnFamily{
 CFName:=self.CName;
 column:=self.simpleattribut->map SAF2Column();column+=self.component->
 collect(j|j.reference)->map reference2Column();
 column+=object Column{CName:="IDRowF"; CType:="uuid";Descriptor:="PrimaryKey";}}}
 --Transformation Component to Row
 mapping Component::component2row():Row{
 Key:=self.CMName;
 ---creation of the column family object as well as transformation of attributes into columns
 columnfamilyR:=object ColumnFamily{CFName:=self.CMName;
 column:=self.simpleattribut->map SAF2Column();column+=self.id->map ID2column();}}
--- transformation - ID To Column
mapping ID::ID2column():Column{CName:=self.IDName;CType:="uuid";Descriptor:="PrimeryKey";}
--Transformation reference for concept to Column
mapping Reference::reference2Column():Column{CName:=self.RFName;CType:=self.RFType;}
--transformation reference for Component To Column
mapping Reference::REF2Column():Column{CName:=self.RFName;CType:=self.RFType;}
--Transformation SimpleAttribut to Column
mapping SimpleAttribut::SAF2Column():Column{CName:=self.SAName;CType:=self.SAType;}
```

(b)



```
modeltype RDBNoSQLGeneric uses "www.RDBNoSQL.com";
modeltype MongoDB_Model uses "www.mongodb.com";
transformation Mongo_normalized(in LogicalModel : RDBNoSQLGeneric , out MongoDB:MongoDB_Model);
main() {LogicalModel.objects()[RDBNoSQL_Schema]->map RDBNoSQLGeneric2MongoDB();}
mapping RDBNoSQL_Schema::RDBNoSQLGeneric2MongoDB():MongoDBModel{
MDName:=self.RNName;
collection:=self.concept->map concept2collection();
collection+=self.concept->collect(i|i.component)->map component2collection();}
--Transformation the concept To Colecction
mapping Concept::concept2collection():Collections{
ClName:=self.CName;atomicfield:=self.simpleattribut->map simpleattribut2atomicfiled();
atomicfield+=self.component->collect(u|u.reference)->map reference2atomicfileid();
id:=object Id{IDName:=self.CName+"_ID";IDType:="UUID";}}
--Transformation the ID To AtomicField
mapping ID::IDC2atomicFileid():AtomicField{
AFName:=self.IDName; AFType:=self.IDType;}
---Transformation the Simple Attribute To Atomic Field
mapping SimpleAttribut::simpleattribut2atomicfiled():AtomicField{
AFName:=self.SAName; AFType:=self.SAType;}
---Transformation the reference To AtomicField
mapping Reference::reference2atomicfileid():AtomicField{
AFName:=self.RFName;AFType:=self.RFType;}
---Transformation Componement To Collections
mapping Component::component2collection():Collections{
ClName:=self.CMName;
atomicfield:=self.simpleattribut->map simpleattribut2atomicFiled();
atomicfield+=self.id->map ID2atomicfileid();
id:=object Id{IDName:=self.CMName+"_ID";IDType:="UUID";}}
---Transformation Simple attribute To Atomic Field
mapping SimpleAttribut::simpleattribut2atomicFiled():AtomicField{
AFName:=self.SAName;AFType:=self.SAType;}
```

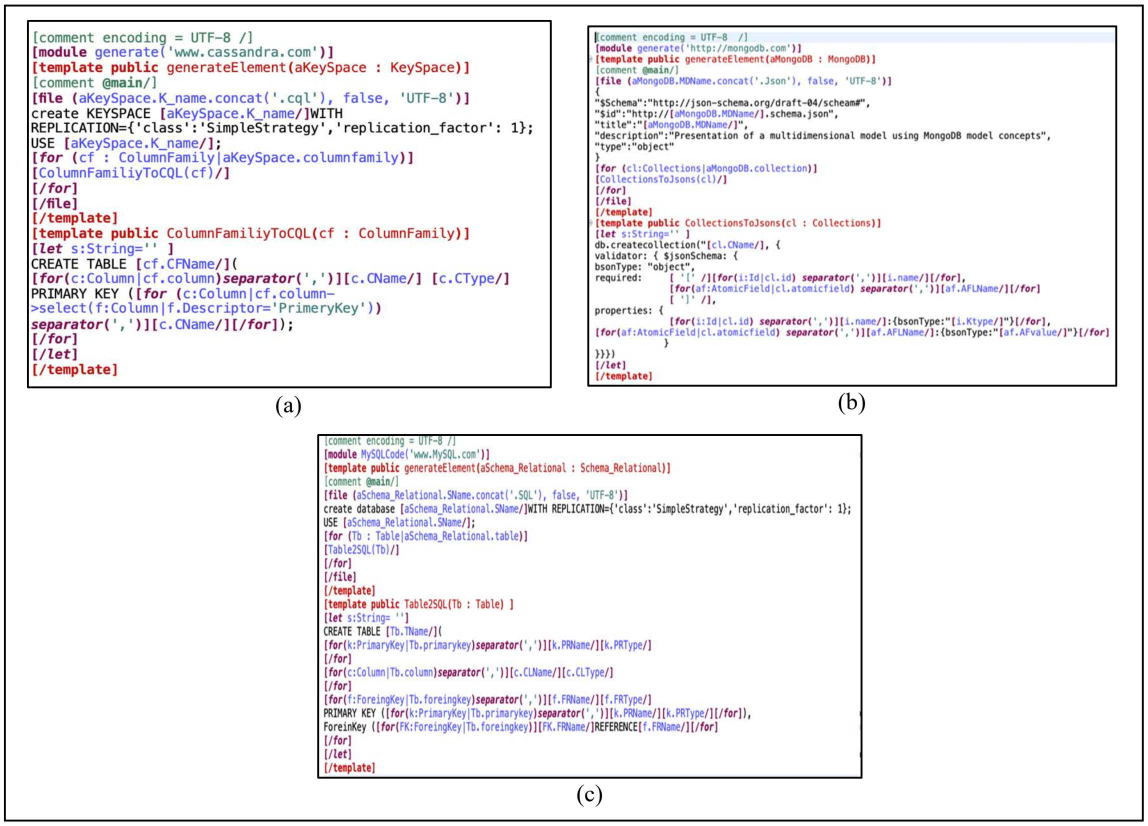
(c)



```
modeltype RDBNoSQLGeneric uses "www.RDBNoSQL.com";
modeltype RelationalModel uses "www.cwm.com";
transformation PIM2RelationalDB(in LogicalModel : RDBNoSQLGeneric , out CWMModel:RelationalModel);
main() {LogicalModel.objects()[RDBNoSQL_Schema]->map RDBNoSQLGeneric2CWMModel();}
mapping RDBNoSQL_Schema::RDBNoSQLGeneric2CWMModel():Schema_Relational
{SName:=self.RNName;table:=self.concept->map Concept2Table();
table+=self.concept->collect(b|b.component)->map Component2Table();}
---Transformation from Concept to Table
mapping Concept::Concept2Table():Table{
TName:=self.CName;primarykey:=self.idC->map IDConcept2PrKey();
column:=self.simpleattribut-> map Simpleattribt2Column();foreingkey:=self.component->
collect(e|e.reference)-> map Reference2FRKey();}
---Transformation from ID to PrimaryKey
mapping ID::IDConcept2PrKey():PrimaryKey
{PRName:=self.IDName;PRType:=self.IDType;}
---Transformation from SimpleAttribute to Column
mapping SimpleAttribut::Simpleattribt2Column():Column
{CLName:=self.SAName;CLType:=self.SAType;}
---Transformation from Reference to ForeinKey
mapping Reference::Reference2FRKey():ForeingKey
{FRName:=self.RFName;FRType:=self.RFType;}
---Transformation from Component to Table
mapping Component::Component2Table():Table
{TName:=self.CMName;primarykey:=self.id->map IdComponent2PRKey();
column:=self.simpleattribut-> map Simpleattribt2Column();}
```

(d)

Figure 9 : QVT Script of the MD2RDBNoSQL transformation (a) and LGM2physicalM (Cassandra, (b) MongoDB, (c) MySQL).



```
[comment encoding = UTF-8 /]
[module generate('www.cassandra.com')]
[template public generateElement(aKeySpace : KeySpace)]
[comment @main/]
[file (aKeySpace.K_name.concat('.cql'), false, 'UTF-8')]
create KEYSPACE [aKeySpace.K_name/]WITH
REPLICATION={'class':'SimpleStrategy','replication_factor': 1};
USE [aKeySpace.K_name/];
[for (cf : ColumnFamily|aKeySpace.columnfamily)]
[ColumnFamiliyToCQL(cf)/]
[/for]
[/file]
[/template]
[template public ColumnFamiliyToCQL(cf : ColumnFamily)]
[let s:String='' ]
CREATE TABLE [cf.CFName/](
[for(c:Column|cf.column)separator(',')][c.CName/] [c.CType/]
PRIMARY KEY ([for (c:Column|cf.column-
>select(f:Column|f.Descriptor='PrimeryKey'))
separator(',')][c.CName/][/for]);
[/for]
[/let]
[/template]
```

(a)

```
[comment encoding = UTF-8  /]
[module generate('http://mongodb.com')]
[template public generateElement(aMongoDB : MongoDB)]
[comment @main/]
[file (aMongoDB.MDName.concat('.Json'), false, 'UTF-8')]
{
"$Schema":"http://json-schema.org/draft-04/scheam#",
"$id":"http://[aMongoDB.MDName/].schema.json",
"title":"[aMongoDB.MDName/]",
"description":"Presentation of a multidimensional model using MongoDB model concepts",
"type":"object"
}
[for (cl:Collections|aMongoDB.collection)]
[CollectionsToJsons(cl)/]
[/for]
[/file]
[/template]
[template public CollectionsToJsons(cl : Collections)]
[let s:String='' ]
db.createcollection("[cl.CName/], {
validator: { $jsonSchema: {
bsonType: "object",
required:      [ '[' /][for(i:Id|cl.id) separator(',')][i.name/][/for],
               [for(af:AtomicField|cl.atomicfield) separator(',')][af.AFLName/][/for]
               [ ']' /],
properties: {
               [for(i:Id|cl.id) separator(',')][i.name/]:{bsonType:"[i.Ktype/]"}[/for],
[for(af:AtomicField|cl.atomicfield) separator(',')][af.AFLName/]:{bsonType:"[af.AFvalue/]"}[/for]
             }
}}})
[/let]
[/template]
```

(b)

```
[comment encoding = UTF-8 /]
[module MySQLCode('www.MySQL.com')]
[template public generateElement(aSchema_Relational : Schema_Relational)]
[comment @main/]
[file (aSchema_Relational.SName.concat('.SQL'), false, 'UTF-8')]
create database [aSchema_Relational.SName/]WITH REPLICATION={'class':'SimpleStrategy','replication_factor': 1};
USE [aSchema_Relational.SName/];
[for (Tb : Table|aSchema_Relational.table)]
[Table2SQL(Tb)/]
[/for]
[/file]
[/template]
[template public Table2SQL(Tb : Table) ]
[let s:String= '']
CREATE TABLE [Tb.TName/](
[for(k:PrimaryKey|Tb.primarykey)separator(',')][k.PRName/][k.PRType/]
[/for]
[for(c:Column|Tb.column)separator(',')][c.CLName/][c.CLType/]
[/for]
[for(f:ForeingKey|Tb.foreingkey)separator(',')][f.FRName/][f.FRType/]
PRIMARY KEY ([for(k:PrimaryKey|Tb.primarykey)separator(',')][k.PRName/][k.PRType/][/for]),
ForeinKey ([for(FK:ForeingKey|Tb.foreingkey)][FK.FRName/]REFERENCE[f.FRName/][/for]
[/for]
[/let]
[/template]
```

(c)

Figure10: MOF2Text translation script for: Cassandra(a), MongoDB((b) and MySQL (c)

4. EXPERIMENTATION

The objective of this section is to describe how the transformations presented in section 3 are used to implement a multidimensional schema at the physical level on two types of systems: NoSQL and relational. We present the steps of the implementation of a case study on a student and courses management application. The objective of this case study is to measure the performance of students according to the results obtained during an academic year on the modules studied as well as the department they are affiliated with. Figure 11 presents the multidimensional model describing this case study.

Thus, we started the first phase of our process by instantiating the PIM conceptual model using the elements of the multidimensional model. In Figure 12(a), the instantiation PIM model is composed of a fact named “Results-Student”, which measures the performance of students by contribution to 4 dimensions, namely, “Student”, “Department”, “Module” and “Date”.

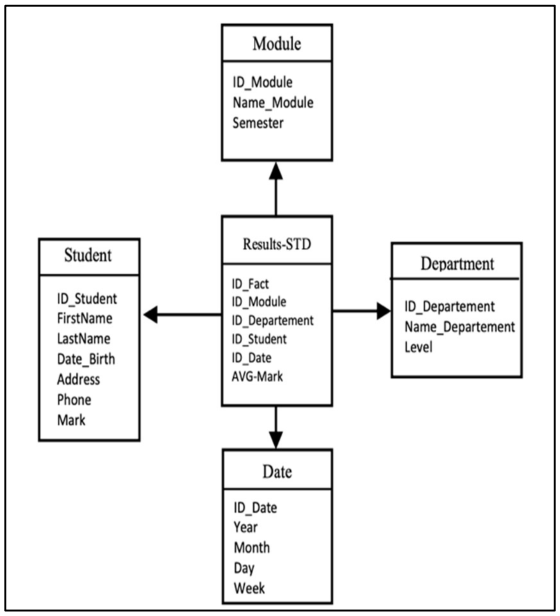
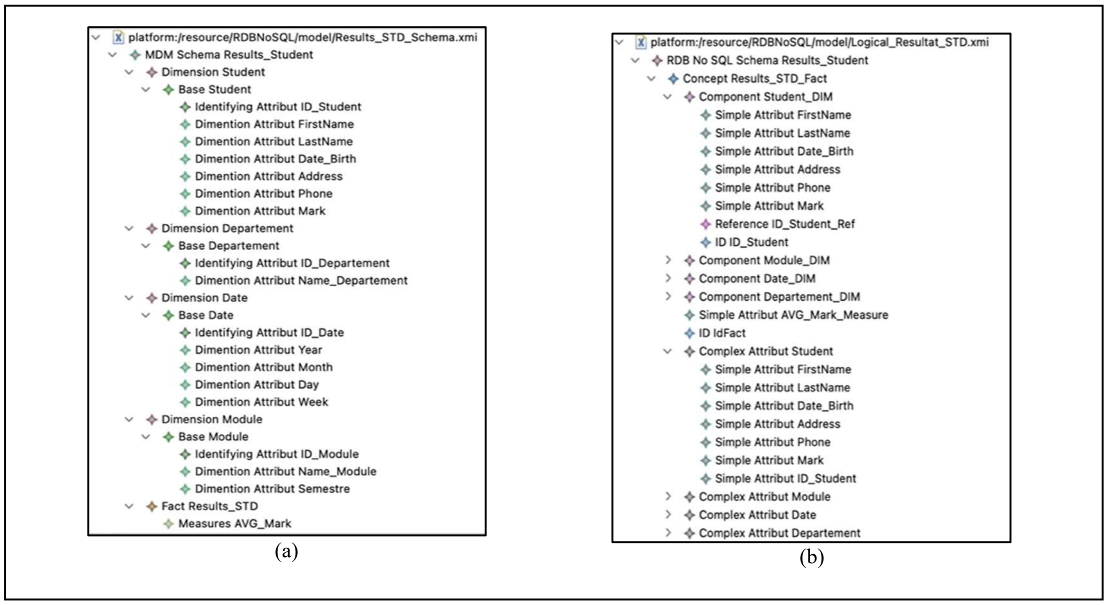


Figure 11: Multidimensional model of the case study

In the second step, the first MD2RDBNoSQL transformation is executed from the instantiated model, which transforms the conceptual model into a generic logical model for both relational and NoSQL systems. The output of this transformation is presented in Fig. 12 (b), hence the logical PIM model is presented in XMI format.

The third step executes the second transformation, which uses the logical PIM model as input to produce 3 types of physical models as output namely, the Cassandra physical model, the MongoDB physical model, and the MySQL physical model. Figure13 ((a) Cassandra, (b) MySQL, (c) MongoDB) presents these models.

Finally, the MOF2Text transformation is executed to translate each physical model into the associated source code. Figure 13 also successively presents the CQL script for the Cassandra model (d), the SQL script for the MySQL model(f), and the JSON Schema script for the MongoDB model(e).



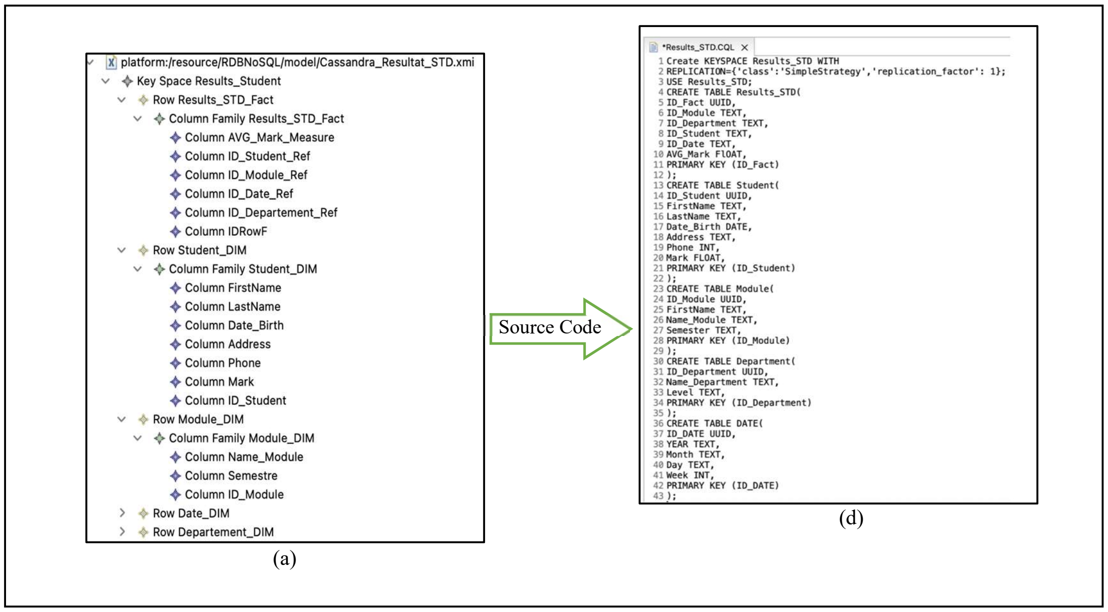
```
platform:/resource/RDBNoSQL/model/Results_STD_Schema.xmi
  MDM Schema Results_Student
    Dimension Student
      Base Student
        Identifying Attribut ID_Student
        Dimention Attribut FirstName
        Dimention Attribut LastName
        Dimention Attribut Date_Birth
        Dimention Attribut Address
        Dimention Attribut Phone
        Dimention Attribut Mark
    Dimension Departement
      Base Departement
        Identifying Attribut ID_Departement
        Dimention Attribut Name_Departement
    Dimension Date
      Base Date
        Identifying Attribut ID_Date
        Dimention Attribut Year
        Dimention Attribut Month
        Dimention Attribut Day
        Dimention Attribut Week
    Dimension Module
      Base Module
        Identifying Attribut ID_Module
        Dimention Attribut Name_Module
        Dimention Attribut Semestre
    Fact Results_STD
      Measures AVG_Mark
```

(a)

```
platform:/resource/RDBNoSQL/model/Logical_Resultat_STD.xmi
  RDB No SQL Schema Results_Student
    Concept Results_STD_Fact
      Component Student_DIM
        Simple Attribut FirstName
        Simple Attribut LastName
        Simple Attribut Date_Birth
        Simple Attribut Address
        Simple Attribut Phone
        Simple Attribut Mark
        Reference ID_Student_Ref
        ID ID_Student
      Component Module_DIM
      Component Date_DIM
      Component Departement_DIM
      Simple Attribut AVG_Mark_Measure
      ID IdFact
      Complex Attribut Student
        Simple Attribut FirstName
        Simple Attribut LastName
        Simple Attribut Date_Birth
        Simple Attribut Address
        Simple Attribut Phone
        Simple Attribut Mark
        Simple Attribut ID_Student
      Complex Attribut Module
      Complex Attribut Date
      Complex Attribut Departement
```

(b)

Figure 12: (a) Multidimensional Results-STD model instantiated by the user ;(b) PIM Logic models generated by our prototype



```
platform:/resource/RDBNoSQL/model/Cassandra_Resultat_STD.xmi
  Key Space Results_Student
    Row Results_STD_Fact
      Column Family Results_STD_Fact
        Column AVG_Mark_Measure
        Column ID_Student_Ref
        Column ID_Module_Ref
        Column ID_Date_Ref
        Column ID_Departement_Ref
        Column IDRowF
    Row Student_DIM
      Column Family Student_DIM
        Column FirstName
        Column LastName
        Column Date_Birth
        Column Address
        Column Phone
        Column Mark
        Column ID_Student
    Row Module_DIM
      Column Family Module_DIM
        Column Name_Module
        Column Semestre
        Column ID_Module
    Row Date_DIM
    Row Departement_DIM
```

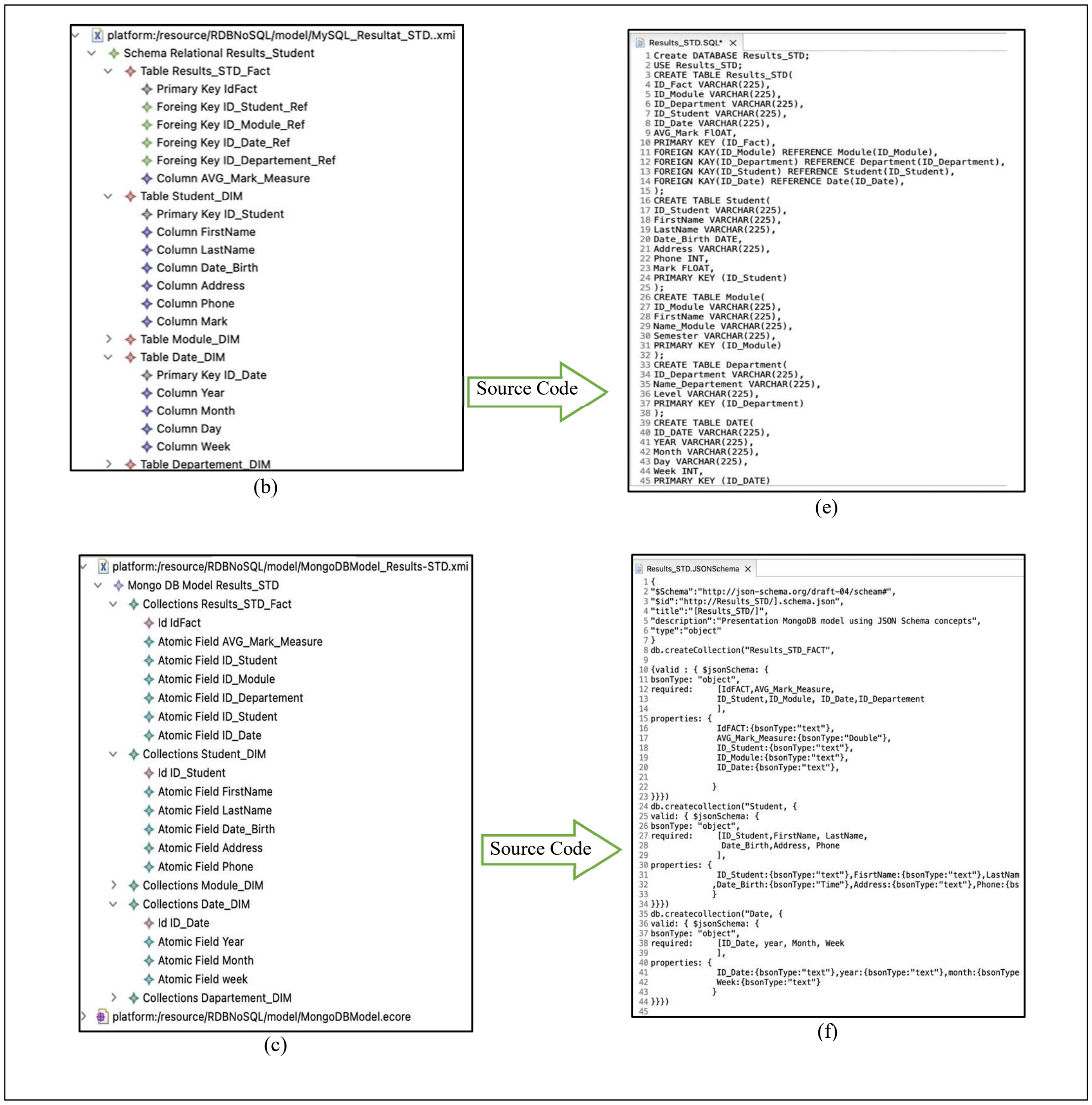
(a)

*Results_STD.CQL

```
Create KEYSPACE Results_STD WITH
REPLICATION={'class':'SimpleStrategy','replication_factor': 1};
USE Results_STD;
CREATE TABLE Results_STD(
ID_Fact UUID,
ID_Module TEXT,
ID_Department TEXT,
ID_Student TEXT,
ID_Date TEXT,
AVG_Mark FlOAT,
PRIMARY KEY (ID_Fact)
);
CREATE TABLE Student(
ID_Student UUID,
FirstName TEXT,
LastName TEXT,
Date_Birth DATE,
Address TEXT,
Phone INT,
Mark FLOAT,
PRIMARY KEY (ID_Student)
);
CREATE TABLE Module(
ID_Module UUID,
FirstName TEXT,
Name_Module TEXT,
Semester TEXT,
PRIMARY KEY (ID_Module)
);
CREATE TABLE Department(
ID_Department UUID,
Name_Department TEXT,
Level TEXT,
PRIMARY KEY (ID_Department)
);
CREATE TABLE DATE(
ID_DATE UUID,
YEAR TEXT,
Month TEXT,
Day TEXT,
Week INT,
PRIMARY KEY (ID_DATE)
);
```

(d)

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195



```
platform:/resource/RDBNoSQL/model/MySQL_Resultat_STD..xmi
  Schema Relational Results_Student
    Table Results_STD_Fact
      Primary Key IdFact
      Foreing Key ID_Student_Ref
      Foreing Key ID_Module_Ref
      Foreing Key ID_Date_Ref
      Foreing Key ID_Departement_Ref
      Column AVG_Mark_Measure
    Table Student_DIM
      Primary Key ID_Student
      Column FirstName
      Column LastName
      Column Date_Birth
      Column Address
      Column Phone
      Column Mark
    Table Module_DIM
    Table Date_DIM
      Primary Key ID_Date
      Column Year
      Column Month
      Column Day
      Column Week
    Table Departement_DIM
```

Results_STD.SQL*

```sql
Create DATABASE Results_STD;
USE Results_STD;
CREATE TABLE Results_STD(
ID_Fact VARCHAR(225),
ID_Module VARCHAR(225),
ID_Department VARCHAR(225),
ID_Student VARCHAR(225),
ID_Date VARCHAR(225),
AVG_Mark FlOAT,
PRIMARY KEY (ID_Fact),
FOREIGN KAY(ID_Module) REFERENCE Module(ID_Module),
FOREIGN KAY(ID_Department) REFERENCE Department(ID_Department),
FOREIGN KAY(ID_Student) REFERENCE Student(ID_Student),
FOREIGN KAY(ID_Date) REFERENCE Date(ID_Date),
);
CREATE TABLE Student(
ID_Student VARCHAR(225),
FirstName VARCHAR(225),
LastName VARCHAR(225),
Date_Birth DATE,
Address VARCHAR(225),
Phone INT,
Mark FLOAT,
PRIMARY KEY (ID_Student)
);
CREATE TABLE Module(
ID_Module VARCHAR(225),
FirstName VARCHAR(225),
Name_Module VARCHAR(225),
Semester VARCHAR(225),
PRIMARY KEY (ID_Module)
);
CREATE TABLE Department(
ID_Department VARCHAR(225),
Name_Departement VARCHAR(225),
Level VARCHAR(225),
PRIMARY KEY (ID_Department)
);
CREATE TABLE DATE(
ID_DATE VARCHAR(225),
YEAR VARCHAR(225),
Month VARCHAR(225),
Day VARCHAR(225),
Week INT,
PRIMARY KEY (ID_DATE)
```

```
platform:/resource/RDBNoSQL/model/MongoDBModel_Results-STD.xmi
  Mongo DB Model Results_STD
    Collections Results_STD_Fact
      Id IdFact
      Atomic Field AVG_Mark_Measure
      Atomic Field ID_Student
      Atomic Field ID_Module
      Atomic Field ID_Departement
      Atomic Field ID_Student
      Atomic Field ID_Date
    Collections Student_DIM
      Id ID_Student
      Atomic Field FirstName
      Atomic Field LastName
      Atomic Field Date_Birth
      Atomic Field Address
      Atomic Field Phone
    Collections Module_DIM
    Collections Date_DIM
      Id ID_Date
      Atomic Field Year
      Atomic Field Month
      Atomic Field week
    Collections Dapartement_DIM
platform:/resource/RDBNoSQL/model/MongoDBModel.ecore
```

Results_STD.JSONSchema

```
{
"$Schema":"http://json-schema.org/draft-04/scheam#",
"$id":"http://Results_STD/].schema.json",
"title":"[Results_STD/]",
"description":"Presentation MongoDB model using JSON Schema concepts",
"type":"object"
}
db.createCollection("Results_STD_FACT",

{valid : { $jsonSchema: {
bsonType: "object",
required:      [IdFACT,AVG_Mark_Measure,
               ID_Student,ID_Module, ID_Date,ID_Departement
               ],
properties: {
               IdFACT:{bsonType:"text"},
               AVG_Mark_Measure:{bsonType:"Double"},
               ID_Student:{bsonType:"text"},
               ID_Module:{bsonType:"text"},
               ID_Date:{bsonType:"text"},

              }
}}})
db.createcollection("Student, {
valid: { $jsonSchema: {
bsonType: "object",
required:      [ID_Student,FirstName, LastName,
                Date_Birth,Address, Phone
               ],
properties: {
               ID_Student:{bsonType:"text"},FisrtName:{bsonType:"text"},LastNam
              ,Date_Birth:{bsonType:"Time"},Address:{bsonType:"text"},Phone:{bs
              }
}}})
db.createcollection("Date, {
valid: { $jsonSchema: {
bsonType: "object",
required:      [ID_Date, year, Month, Week
               ],
properties: {
               ID_Date:{bsonType:"text"},year:{bsonType:"text"},month:{bsonType
               Week:{bsonType:"text"}
              }
}}})
```

Figure 13: Physical models generated from logical PIM and translated into associated source code: (a)PSM Cassandra: (d) CQL script;(b) PSM MySQL: (e) SQL script; (c)PSM MongoDB: (f)JsonSchema Script.

5. EVALUATION APPROACH:

In the present section, we will evaluate the hybrid approach proposed in this article for implementing a data warehouse on two NoSQL and relational systems.

More specifically, we will carry out a qualitative comparison between our prototype and the works presented in section 2.

To establish this comparison and come up with reliable results, only the works that are similar to our approach are included, namely, works [7,10,12,16,13-14].

Table 1 presents our qualitative comparison according to the following four criteria:

- ✓ The use of the multidimensional model (MD) in transformations (1),
- ✓ The level of modeling (2) (C: conceptual; L: logical; P: Physical),
- ✓ Compatibility with 3 NoSQL systems (3) (CL: Column; DC: Document, GR: Graph; RD: Relational),
- ✓ Automation of transformations (4).

Table 1: Comparative Study between our prototype and related works

Criteria/ Works	(1)	(2)			(3)				(4)
		C	L	P	CL	DC	GR	RD	
[10]	x	x		x	x				-
[12]	x	x		x			x		-
[16]	x	x	x			x			-
[7]	x	x		x				x	-
[13]	-		x		x	x	x	x	x
[14]	-	x	x		x	x	x	x	x
Our Approach	x	x	x	x	x	x	x	x	x

From this table, we can see that these works are similar to our approach, especially at the level of the following aspects:

- ✓ The multidimensional model represents the conceptual level for the design of a data warehouse encompassing NoSQL and relational systems. It is composed of various elements, namely facts, dimensions, measures, and hierarchies, which are then transformed into the target components. This is represented at the conceptual level in the works [10,12,7,16] in addition to our approach.
- ✓ Compatibility with NoSQL and relational systems is addressed in our approach, as well as in the works [13,14], through the use of a generic logical model which is implemented at the physical level on four families of Database Management Systems (DBMS): column-oriented, document-oriented, graph-oriented, and relational databases.
- ✓ The correspondence between the models is obtained through the implementation of transformation rules, thus simplifying the automatic conversion of the conceptual model structures into a logical and physical model. This automation is demonstrated by the MDA approach in our prototype, as well as in the work [14].

There are also notable distinctions between our proposal and the existing works, including:

- ✓ Our approach includes the 3 levels of abstraction for designing a data warehouse, namely the conceptual, logical, and physical levels. On the other hand, the body of work only covers two levels, namely the conceptual and physical level, or the logical and physical level.
- ✓ For the work [16], it involves document-oriented systems only at the logical level, which does not ensure the sustainability of the logical model, which is a crucial aspect of the MDA approach. In our approach, this aspect of sustainability is ensured since our generical model can describe all NoSQL and relational families.
- ✓ In our approach, as in the work [13-14], automation is based on the MDA methodology. However, the transformation to the physical level is not covered in those works, unlike our approach which not only ensures this transition but also makes it possible to extract the source code from each physical model, namely, the Cassandra model, MongoDB, and MySQL.

Based on this qualitative comparison, we recognize that our hybrid approach to data warehouse design addresses all the limitations presented above. Our approach proposes an architecture with three levels of abstraction, namely, conceptual, logical, and physical. The conceptual level is represented by a multidimensional model consistent with data warehousing concepts. The logical level is implemented in a generic manner, adaptable to the three families of NoSQL systems as well as relational systems. Finally, the physical model is described according to three DBMS models, specifically the MySQL model, Cassandra model and MongoDB model. Concerning the transitions between these levels, the MDA approach is utilized to automate the transition between these models and generate the source code. It should be noted that the logical PIM can also describe the other two NoSQL families, key-value oriented and Neo4J, but due to space constraints, we limit our discussion to the NoSQL families mentioned above.

In addition, our approach has some limitations, especially compared to previous work [13-14], which explored the concept of polyglot persistence in the design of the generic logic model. Polyglot persistence implies the use of a single multi-model database management system (DBMS) capable of integrating both NoSQL and relational systems. Unlike our approach, which uses separate DBMSs, polyglot persistence offers a consolidated solution.

The difficulty with our approach lies in the use of separate DBMSs for NoSQL and relational systems. This creates complexities in terms of system management, integration and maintenance. Using separate DBMSs can result in potential problems such as data inconsistency, increased operating costs and the requirement for specialist expertise to manage different database technologies.

On the other hand, the main advantage of polyglot persistence lies in its ability to integrate different types of databases within a single system, facilitating the handling of diverse data requirements. By taking advantage of this approach, businesses can benefit from a single DBMS to meet the different storage needs of NoSQL and relational data, rationalizing the architecture, reducing operational costs and improving efficiency.

Despite these advantages, implementing polyglot persistence can be complex and costly. The technical complexities involved in managing different data structures in a single system require extensive expertise and significant resources. What's more, while this approach may be beneficial for some companies, it is not necessarily accessible to all, especially those that have already had functional data warehouses in place for years.

Ultimately, the choice between an approach using distinct DBMSs and polyglot persistence depends on the specific needs of each organization, considering the complexity of implementation, associated costs, and the ability to effectively manage data diversity.

6. CONCLUSION AND FUTURE WORK

The work presented in this article falls into the category of decision support systems. We have developed a model-driven approach to designing a hybrid storage architecture to overcome the limitations of data warehouses in the face of large-scale data. To do this, we have established a set of transformation rules that automate the creation of a data warehouse, starting with the implementation of the multidimensional conceptual model into a generic logical model adapted to NoSQL and relational databases. This model is then implemented at the physical level on three different DBMS platforms: MongoDB, MySQL and Cassandra. The transition between models is made by the QVT language. Finally, from each physical model, we used the MOF2Text language to extract the appropriate creation source code for each DBMS.

In our forthcoming research, our objective is to develop a method focused on automating the extraction of a multidimensional model from a substantial data source. This data source represents a consolidation area for the data warehouse, which is used to integrate the information system of a specific organization. This novel approach will subsequently be integrated with the one presented in this article, thereby establishing a comprehensive automated process, ranging from the integration of data sources to the generation of design source code for the data warehouse.

AUTHOR CONTRIBUTIONS.

Lamya oukhouya : Conceptualization; methodology; writing-original draft preparation. **Anass El Haddadi** and **Brahim ER-Raha :** Supervision and validation. **Asma Sbai** : review and editing. All authors read and agreed to the published version of the manuscript.

REFERENCES:

[1] Imam, A. A., Basri, S., Ahmad, R., Watada, J., González Aparicio, M. T., & Almomani, M. A. (2018). Data modeling guidelines for NoSQL document-store databases. International Journal of Advanced Computer Science and Applications, 9.

[2] Oukhouya, L., El haddadi, A., Er-raha, B., Asri, H., & Sbai, A. (2022, May). Designing Hybrid Storage Architectures with RDBMS and NoSQL Systems: A Survey. In International Conference on Advanced Intelligent Systems for Sustainable Development (pp. 332-343). Cham: Springer Nature Switzerland.

[3] El Mouden, Z. A., & Jakimi, A. (2020). A New Algorithm for Storing and Migrating Data Modelled by Graphs.

[4] Padhy, R. P., Patra, M. R., & Satapathy, S. C. (2011). RDBMS to NoSQL: reviewing some next-generation non-relational

database's. International Journal of Advanced Engineering Science and Technologies, 11(1), 15-30.

[5] El Beggar, O., Letrache, K., & Ramdani, M. (2021). DAREF: MDA framework for modelling data warehouse requirements and deducing the multidimensional schema. Requirements Engineering, 26(2), 143-165.

[6] Hanine, M., Lachgar, M., Elmahfoudi, S., & Boutkhoum, O. (2021). MDA Approach for Designing and Developing Data Warehouses: A Systematic Review & Proposal. International Journal of Online & Biomedical Engineering, 17(10)

[7] Aldisa, R. T., & Abdullah, M. A. (2022). Implementation of Data Warehouse for Food Sales Strategy Using Snowflake Schema Model. IJISTECH (International Journal of Information System and Technology), 6(2), 254-258.

[8] Nyunt, L. S. T. Implementation of Multidimensional Data Models using Data Warehouse and OLAP Technology (Doctoral dissertation, MERAL Portal).

[9] Batoure, A. B. Using Model-Driven Engineering for Decision Support Systems Modelling, Implementation and Powering.

[10] Khalil, A., & Belaissaoui, M. (2023). An Approach for Implementing Online Analytical Processing Systems under Column-Family Databases. IAENG International Journal of Applied Mathematics, 53(1).

[11] Khalil, A., & Belaïssaoui, M. (2020, December). Key-value data warehouse: Models and OLAP analysis. In 2020 IEEE 2nd International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS) (pp. 1-6). IEEE.

[12] Sellami, A., Nabli, A., & Gargouri, F. (2020, November). Graph NoSQL data warehouse creation. In Proceedings of the 22nd International Conference on Information Integration and Web-based Applications & Services (pp. 34-38).

[13] Mali, J., Atigui, F., Azough, A., & Travers, N. (2020). ModelDrivenGuide: an approach for implementing NoSQL schemas. In Database and Expert Systems Applications: 31st International Conference, DEXA 2020, Bratislava, Slovakia, September 14–17, 2020, Proceedings, Part I 31 (pp. 141-151). Springer International Publishing.

[14] Candel, C. J. F., Ruiz, D. S., & García-Molina, J. J. (2022). A unified metamodel for NoSQL and relational databases. Information Systems, 104, 101898.

[15] Pokorny, J. (2018). Integration of relational and graph databases functionally. arXiv preprint arXiv:1809.03822.

[16] Chavalier, M., El Malki, M., Kopliku, A., Teste, O., & Tournier, R. (2016, June). Document-oriented data warehouses: Models and extended cuboids, extended cuboids in oriented document. In 2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS) (pp. 1-11). IEEE.

[17] Oukhouya, L., Er-raha, B., & Asri, H. (2021). A generic metadata management model for heterogeneous sources in a data warehouse. In E3S Web of Conferences (Vol. 297). EDP Sciences