UMLto[No]SQL: Mapping Conceptual Schemas to Heterogeneous Datastores

Gwendal Daniel
Internet Interdisciplinary Institute, IN3
Universitat Oberta de Catalunya, UOC
Barcelona, Spain
gdaniel@uoc.edu

Abel Gómez
Internet Interdisciplinary Institute, IN3
Universitat Oberta de Catalunya, UOC
Barcelona, Spain
agomezlla@uoc.edu

Jordi Cabot
ICREA
Universitat Oberta de Catalunya, UOC
Barcelona, Spain
jordi.cabot@icrea.cat

***Abstract*—The growing need to store and manipulate large volumes of data has led to the blossoming of various families of data storage solutions. Software modelers can benefit from this growing diversity to improve critical parts of their applications, using a combination of different databases to store the data based on access, availability, and performance requirements. However, while the mapping of conceptual schemas to relational databases is a well-studied field of research, there are few works that target the role of conceptual modeling in a multiple and diverse data storage settings. This is particularly true when dealing with the mapping of constraints in the conceptual schema. In this paper we present the UMLto[No]SQL approach that maps conceptual schemas expressed in UML/OCL into a set of logical schemas (either relational or NoSQL ones) to be used to store the application data according to the data partition envisaged by the designer. Our mapping covers as well the database queries required to implement and check the model's constraints. UMLto[No]SQL takes care of integrating the different data storages, and provides a modeling layer that enables a transparent manipulation of the data using conceptual level information.**

***Index Terms*—Database Design, UML, OCL, NoSQL, RDBMS, constraint, SQL, model partitioning**

I. Introduction

The NoSQL movement [6] is recognized as one of the main solutions to handle large volumes of diverse data. NoSQL solutions are based on a flexible schemaless data model focused on efficiency, horizontal scalability, and high availability, and can be categorized into four major groups: key-value stores, document databases, graphs, and column databases [22]. NoSQL stores complement relational databases and other hybrid solutions to offer a large number of options when it comes to decide where to store the data managed by a software system. In fact, it is not unusual that the best alternative is to combine several technologies for an optimal result. E.g., the same application could use a relational database for transactional data but opt for a key-value store for event log management.

Unfortunately, current conceptual modeling methods and tools offer little support for this "multi-store" modeling and generation challenge. Modelers should be able to easily annotate the model with details on the type of data solution to use as storage mechanism and to generate from that annotated model the corresponding implementation in a (semi)automatic way. Still, only the mapping of conceptual schemas to relational databases has been widely investigated so far. Very few solutions targeting NoSQL backends exist and, when they do, they target individual datastores and typically skip the mapping of the model constraints.

In this paper, we present UMLto[No]SQL, a new approach to partition conceptual schemas (including constraints and business rules) and to map each fragment to a different data storage solution. Without loss of generality we assume that the conceptual schema is modeled as a UML class diagram [25] and its integrity constraints are expressed in OCL [24] (Object Constraint Language), both OMG standards.

UMLto[No]SQL is based on existing works targeting the mapping of conceptual schemas to specific datastore technologies [9], [11] and integrates them in a unique multi-datastore design method. The main contributions of the paper are:

- A conceptual model partitioning approach integrated as an UML profile allowing to group model elements in regions to be persisted in a specific datastore.
- A set of mappings from conceptual schema regions to datastore-specific models, including a new mapping from UML class diagrams to document databases.
- A translation approach from constraints and business rules to database-specific queries, also covering constraint expressions that require evaluating elements stored in different and heterogeneous backends.
- A runtime modeling layer that automatically deploys the modeled application, and provides a high-level API to manipulate and check constraints on top of the underlying persistent storage mechanisms transparently.

The rest of the paper is organized as follows: Section II presents the UMLto[No]SQL approach, Section III introduces our model partitioning technique, Section IV details the mapping of (UML) models to a variety of store metamodels and Section V does the same for the OCL constraints. Section VI introduces the UMLto[No]SQL runtime layer. Finally, Section VII describes the related works, and Section VIII summarizes the contributions and draws future work.

II. UMLto[No]SQL Approach

The MDA standard [23] proposes a structured methodology to system development that promotes the separation between a platform independent specification (Platform Independent

Model, PIM), and the refinement of that specification (Platform Specific Model, PSM) adapted to the technical constraints of the implementation platform. In this context, a model-to-model transformation (M2M) generates the PSMs from the PIMs, while a model-to-text transformation (M2T) typically takes care of producing the final system implementation out of the PSM models. This PIM-to-PSM phased architecture brings two important benefits: *(i)* the PIM level focuses on the specification of the structure and functions, raising the level of abstraction and postponing technical details to the PSM level; and *(ii)* multiple PSMs can be generated from one PIM, improving portability and reusability.

This PIM-to-PSM staged development perfectly fits the phases of our UMLto[No]SQL approach, where a runtime modeling layer – able to interact with heterogeneous datastores – is generated out of a conceptual schema expressed in UML and OCL. Thus, following the MDA approach, Fig. 1 describes a typical UMLto[No]SQL development process together with the main artifacts involved.

The only actor involved in a UMLto[No]SQL process is the system modeler, who specifies the conceptual schema of the system using a *UML class diagram* and its corresponding *OCL constraints* at the PIM level. At this level, the modeler can manually annotate *i* the conceptual schema (see Section III-B for more details) to group its entities in different *regions*. A *region* is a logical partition that determines the conceptual elements that should be persisted in the same datastore. For example, the *Annotated UML class diagram* in Fig. 1 has been annotated specifying two regions: one contains the elements with horizontal stripes, and the other contains the elements with vertical stripes. The initial OCL expressions remain unaffected.

Next, two automatic M2M transformations are executed in parallel to convert our PIMs into PSMs:

- **UmlTo[No]SQL** *ii* transforms each region – and its elements – of the annotated model at the PIM level, into a set of datastore-specific models at the PSM level (the so-called (*Datastore Models*), precisely representing the data structures to be used to record the data in that particular datastore. In Fig. 1, elements annotated with vertical stripes are transformed to a *Datastore model* for the ○ backend, while elements annotated with horizontal stripes are transformed to a *Datastore model* for the ⬠ backend. Note that this transformation also handles the translation of relationships among elements in different regions producing cross-*Datastore Models* relationships (⇔). These special relationships are discussed in Section IV.
- **OCLToQuery** *iii* transforms the OCL constraints at the PIM level into a set of *Query Models* adapted to the selected storage technology. As it can be seen in Fig. 1, *Query Models* can contain both elements of the ○ and the ⬠ backend. This is because the transformation also handles the translation of queries spanning different datastores. Such cross-datastore queries exploit a dedicated query mechanism presented in Section V.

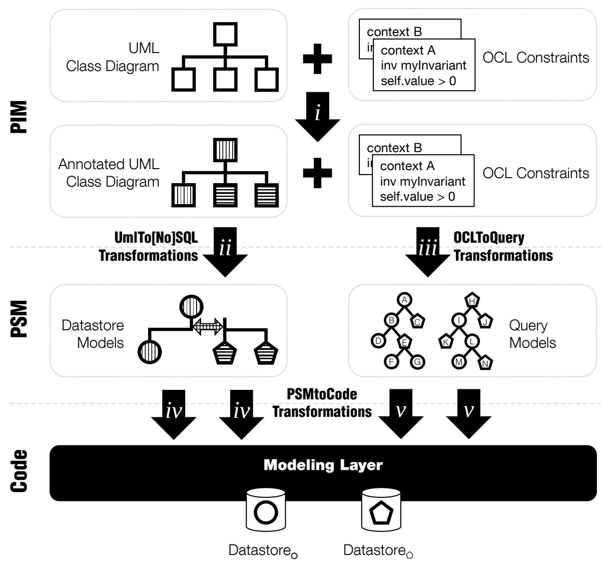

Fig. 1. Overview of the UMLto[No]SQL Infrastructure

Finally, the generated *Datastore Models* and *Query Models* are sent to a final set of *PSMtoCode* M2T transformations. These transformations generate the database schema definitions (in the case of SQL datastores) and software artifacts (in all cases) that are capable of retrieving the different entities of the conceptual schema from their corresponding datastores *iv*. Additionally, the code to evaluate the different queries and constraints is also generated *v*.

The result of these transformations is an integrated *Modeling Layer* that allows interacting with the different databases, retrieving and modifying the data stored in them, and also checking the corresponding integrity constraints. All accesses to the underlying data are done in a transparent way, only using conceptual level information. This *Modeling Layer* can be seen as a runtime model [2] built on top of the storage solutions. That way, users can query the data by directly referring to the types and associations they are interested in, instead of having to know the internal data structure names used to record that data in each specific datastore.

III. Conceptual Schema Partitioning

As introduced in the previous Section, the first step of our approach is partitioning the initial conceptual schema into a set of *regions*. Each region identifies a group of entity types in the schema that should be stored together in the datastore associated to the region. A schema can have one or more regions. The regions of a schema can be homogeneous (all regions are linked to a different backends but all backends are of the same "family") or heterogeneous (different regions may be linked to completely different types of backends).

At the PIM level, the modeler only needs to focus on defining the limits of the region and the desired type of datastore. Low-level details (e. g., connection information) are provided later on, in the PSM models, as part of the typical MDA refinement process (see Section IV).

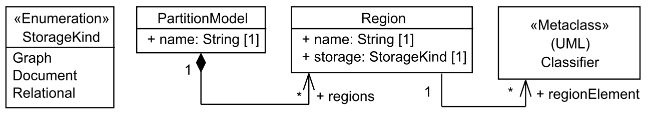

Fig. 2. Partition Domain Model

Without loss of generality, conceptual schemas are expressed as UML class diagrams but other conceptual modeling languages could have been used instead. Still, UML offers a modeling primitive that is especially useful to implement our concept of regions, the UML *profiling mechanism*.

Profiling opens the possibility of extending or restricting UML to satisfy the modeler needs in a standard way. Moreover, the vast majority of UML tools support this mechanism, facilitating a quick adoption of profiles like ours.

To construct a technically correct high-quality UML profile, several steps need to be followed according to the generally accepted good practices [27]. First, a *domain model* defining the additional modeling concepts must be created. In our case, this domain model should capture the concepts to specify schema partitions as presented next in Sect III-A. Second, a *profile* is defined by mapping the concepts from the domain model to UML concepts. In our case, each class of the partition domain model is examined, together with its attributes, associations and constraints, to identify the most suitable UML base concepts for it. Based on this, we built the partition profile presented in Sect. III-B.

A. A domain model for conceptual schemas partitioning

Figure 2 shows our proposed domain model for the partitioning of conceptual schemas. In this proposal, a given conceptual schema (not shown) relates to one *Partition Model* containing the *Regions* for the different datastores the modeler wants to use. *Regions* have a *name* that identifies them, and a *storage* attribute indicating the type of data storage approach that will be used in subsequent steps. Thee possible data storage approaches can be specified, each one associated to a different data persistence technique: *Graph* for graph-based databases, *Document* for document-based stores, and *Relational* for classical *relational* databases. Finally, UML classes from the conceptual schema may be associated to a specific region via its *regionElement* association. All UML classes belonging to the same region should be stored together at the end of the development process.

B. A profile for conceptual schemas partitioning

In order to put our partition approach in practice, we have defined the accompanying UML profile as proposed by Selic [27]. Following the good practices proposed by Lagarde [18], Figure 3 shows our UML profile for conceptual schemas partitioning.

As it can be seen, there exists almost a one-to-one mapping between the concepts of the *Partition Domain Model* and the *Partition Profile*. Now, the link between a *Partition Model* and a conceptual schema represented as a UML *Model* is explicit via the *extension* relationship. For its part, *Regions* extend *Namespaces*, which typically, are *Packages* containing the entity types (e. g., classes) of the conceptual schema. Finally, the *RegionElement* stereotype is used to annotate the *Classifiers* (typically UML *classes*) within a *Region* that will be persisted.

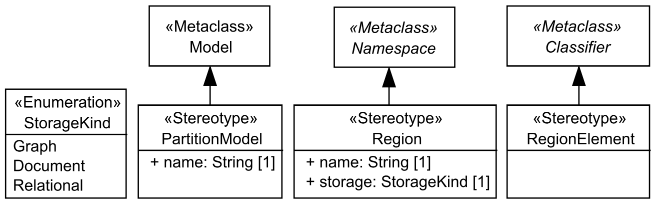

Fig. 3. Partition Profile

The *Partition Profile* is exemplified in Figure 4. The conceptual schema shown in the figure depicts an e-commerce application that will be used as running example for the rest of the paper. The schema specifies the *Client*, *Orders*, *Products*, and *Comment* concepts and the relationships among them, organized in three packages *Clients*, *Business*, and *Comments*.

By means of the profile, we have expressed that the *business* part of this schema should be stored in a document database, the *clients* one should go to a relational database, and the *comments* one to a graph database. Therefore, we have defined three *Regions*, each one tagged with a different

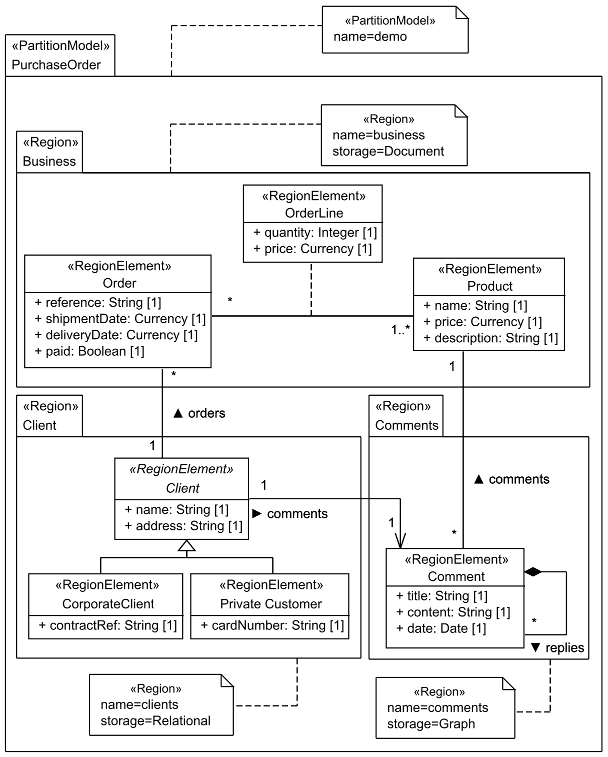

Fig. 4. Example Schema Partition

target storage solution. Note that in our example *Regions* are created based on the domains of their data, but alternative *Partition Models* based on performance requirements (e.g. data proximity) can also be specified through our profile

IV. Mapping Class Diagram Fragments

The partitioned conceptual schema is the input of our UMLto[No]SQL transformation component, in charge of generating the PSMs representing the logical database structure corresponding to each conceptual region in the partition.

Our approach currently embeds three transformations, each one associated to a specific data store. Two of them are based on previous works: *UmlToGraphDB* covering mappings to graph databases (taken from [9]) and *UmlToSQL* that reuses “classical” works on mapping UML (or ER) to relational databases (e.g. [11]). The third one is completely new. As it is also a novel contribution the possibility to have in the same schema regions linked to heterogeneous backends. As such, in this section we review the relevant parts of existing UML to database mappings and show how we adapt them to handle cross-datastore associations. We also introduce *DocumentDB*: a novel metamodel representing the structure of Document databases, as well as the associated mapping rules to derive DocumentDB instances from class diagrams.

In the following we denote a partitioned class diagram *CD* as a tuple $CD = (Cl, As, Ac, I, R)$, where *Cl* is the set of classes, *As* is the set of associations, and *Ac* is the set of association classes. *I* is a set of pairs of classes such as (cl_1, cl_2) represents the fact that c_1 is a direct or indirect subclass of c_2, and *R* is a set of pairs (cl, r) mapping each class of the conceptual model to a region defined with the help of our partition metamodel. Note that for the sake of simplicity attributes of classes and associations are denoted $cl.attr$ (e.g. $cl.name$ represents the name of a class).

We also define two additional predicates that are reused across our mappings: $reg(cl) = r.storage, (cl, r) \in R$ represents the *storage* value of the region containing the class *cl*, and $parents(c) \subset Cl, \forall p \in parents(c), (c, p) \in I$ represents the set of super-classes in the hierarchy of *c*.

A. Relational Database Mapping

1) RelationalDB Metamodel: Figure 5 presents our RelationalDB metamodel which is a simplified version of existing metamodels aiming at representing relational databases. It contains a top-level *Schema* that stores a set of named *Tables*. *Tables* contain *Columns* that store information of a given *DataType*. Our metamodel currently supports primitive types, as well as an *UUID* type which is used to identify elements across datastores. A *Table* also contains a *PrimaryKey* which refers to one or multiple *Columns*. Finally, *Tables* can contain *ForeignKeys*, each one associated to an existing *PrimaryKey* and linking a *Column* to a foreign one stored in another *Table*[1].

[1]For simplicity purposes, this metamodel does not support foreign keys spanning over multiple columns

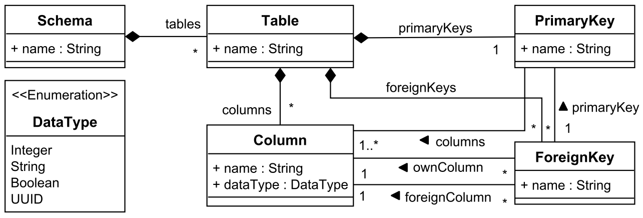

Fig. 5. RelationalDB Metamodel

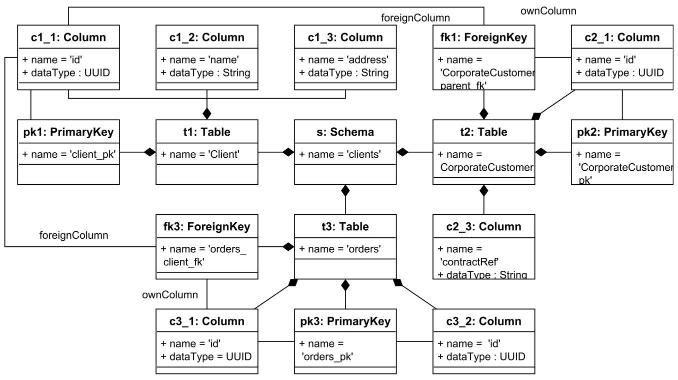

Fig. 6. RelationalDB Instance

2) Class Diagram to RelationalDB Transformation: We denote a RelationalDB model *RM* as a tuple $RM = (S, T, C, P, F)$, where *S* is a schema, *T* is the set of tables, *C* is the set of columns, *P* is the set of primary keys, and *F* is the set of foreign keys that compose the relational model. In the following we present the mapping from *CD* to *RM*. This mapping is based on existing approaches from the literature [11], and adapts them to support cross-datastore queries.

- **R1:** each relational region $r.storage = Relational$, $\forall cl(cl, r) \in R$ is mapped to a schema s such as $s.name = r.name$.
- **R2:** each class $cl \in Cl$ is mapped to a table $t \in T$, where $t.name = cl.name$, and added to the schema s mapped from its containing region such as $t \in s.tables$. In addition, a column $c \in C$ is created such as $c.name = id$, $c.dataType = UUID$, $c \in t.columns$. This column is set as the primary key pk of t such as $pk \in P$, $pk.name = cl.name + \_pk$, $c \in pk.columns$, $pk \in t.primaryKey$.
- **R3:** each inheritance relationship between two classes cl_1, $cl_2 \in Cl$, $(cl_1, cl_2) \in I$ (respectively mapped by R2 to t_1, $t_2 \in T$) such as $reg(cl_1) = reg(cl_2) = Relational$ is mapped to a foreign key such as $fk \in F$, $fk.name = cl_1.name\_parent\_fk$, $fk \in t_1.foreignKeys$ such as $fk.primaryKey = t_2.primaryKey$, $fk.ownColumn = t_1.primaryKey.column$, $fk.foreignColumn = t_2.primaryKey.column$. Note that this rule does not create foreign keys for cross-datastore inheritance links.
- **R4:** each attribute of a class $cl \in Cl$ such as $a \in cl.attributes$ is mapped to a column $c \in C$ where

$c.name$ = $a.name$, $c.datatype$ = $a.type$[2], and added to the column list of its mapped container t such as $c \in t.columns$.

- **R5:** each single-valued association $as_{single} \in As$ between two classes $cl_1, cl_2 \in Cl$ is mapped to a column $c \in C$, $c.name = as.name$, $c.dataType = UUID$ added to the table t_1 such as $c \in t_1.columns$. If $reg(c_1) = reg(c_2) = Relational$, a foreign key $fk \in F$ is also created such as $fk.name = as_{single}.name + \_ + cl_2.name + \_fk$, $fk.primaryKey = t_2.primaryKey$, $f.ownColumn = c$, $f.foreignColumn = t_2.primaryKey.column$ and added to the containing table's foreign key set such as $f \in t_1.foreignKeys$. This means that cross-datastore associations are mapped to UUID columns (with no corresponding foreign key).
- **R6:** each multi-valued association $as_{multi} \in As$ between two classes cl_1, $cl_2 \in Cl$ is mapped to a table $t \in T$, $t.name = as.name$ containing two columns c_1, $c_2 \in C$, c_1, $c_2 \in t.columns$ such as $c_1.name = cl_1.name + \_id, c_1.dataType = UUID$, $c_2.name = cl_2.name + \_id, c_2.dataType = UUID$. This mapping rule also creates a primary key $pk \in P$, $pk.name = as_{multi}.name\_pk$, $pk \in t.primaryKeys$ such as c_1, $c_2 \in pk.columns$. Finally, classes involved in the association such as $reg(cl_n) = Relational$ are mapped to a foreign key $fk \in F$, $fk.name = as_{multi}.name + \_cl_n.name + \_fk, fk \in t.foreignKeys$ such as $fk.primaryKey = t_n.primaryKey$, $fk.ownColumn = c_n$, $fk.foreignColumn = t_n.primaryKey.column$.

Figure 6 presents the RelationalDB model produced when applying the mapping rules to our running example. The produced model defines a top-level *Schema* element named *clients* and containing three *Tables*. The *Table* `t1` is mapped from the *Client* abstract class, and contains three columns to store the *Client*'s UUID, name, and address. It also defines a *PrimaryKey* `pk1` defined over the UUID column. The *Table* `t2` is mapped from the *CorporateCustomer* class, and defines two columns to store its UUID and its contractRef attributes. It also define a *PrimaryKey* `pk2` over its id column, and a *ForeignKey* `fk1` associating the *CorporateCustomer id* to its parent *Client* one. Finally, `t3` is created from the *order_client* association, and contains two *Columns* (set as primary key through the `pk3` instance) storing the identifiers of the involved *Client* and *Order* instances. This *Table* contains a single *ForeignKey* linking the *client_id Column* to the *id Column* of the *Client* table. Since *Order* is not part of the relational region, the mapping does not produce a foreign key for this column. Navigation of this cross-datastore association is detailed in Section V.

B. Graph Database Mapping

1) GraphDB Metamodel: Figure 7 presents the GraphDB metamodel [9] that defines the possible strutural elements

[2]We assume that a mapping mechanism is available to convert conceptual model types to the ones defined in our target metamodels.

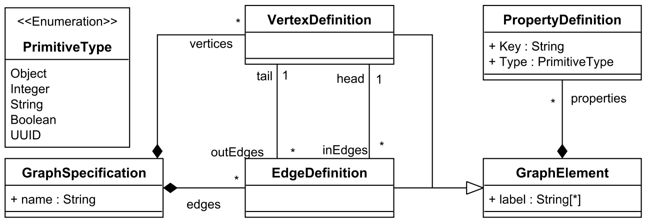

Fig. 7. GraphDB Metamodel

in graph databases. This metamodel is compliant with the Blueprints specification [28], which is an interface designed to unify graph databases under a common API.

The GraphDB metamodel defines a top-level *GraphSpecification* which contains all the *VertexDefinitions* and *EdgeDefinition* of the database under design. *VertexDefinitions* and *EdgeDefinitions* can be linked together using *outEdges* and *inEdges* association, meaning respectively that a *VertexDefinition* has outgoing edges and incoming edges. In addition, *VertexDefinition* and *EdgeDefinition* are both subtypes of *GraphElement*, which defines a set of *labels* describing the type of the element, and a set of *PropertyDefinition* through its *properties* association. In graph databases properties are represented by a *key* (the name of the property) and a *type*. The GraphDB metamodel defines 5 primitives types (Object, Integer, String, Boolean, and our own UUID type).

2) Class Diagram to GraphDB Transformation: Herein we present the mapping from a partitioned class diagram to the GraphDB metamodel. This mapping is an adaptation of our previous work [9].

We denote a GraphDB model *GD* as a tuple $GD = (G, V, E, P)$, where *G* is a *GraphSpecification*, *V* is set of *VertexDefinitions*, *E* the set of *EdgeDefinitions*, and *P* the set of *PropertyDefinitions* that compose the graph. The mapping from *CD* to *GD* is formalized in the following rules:

- **R1:** each graph region $r.storage = Graph, \forall cl(cl, r) \in R$ is mapped to a graph specification g such as $g.name = r.name$.
- **R2:** each class $cl \in Cl, \neg cl.isAbstract$ is mapped to a vertex definition $v \in V$, where $v.label = cl.name \cup parents(cl).name$. In addition, a property $p \in P, p.key = id, p.type = UUID$ is added to the property list of the created vertex such as $p \in v.properties$. This property is used to uniquely identify the element and will be used to retrieve specific instances when navigating cross-datastore associations.
- **R3:** each attribute $a \in (cl \cup parents(cl)).attributes$ is mapped to a property definition p, where $p.key = a.name$, $p.type = a.type$, and added to the property list of its mapped container v such as $p \in v.properties$. Note that this rule flattens the inheritance hierarchy by mapping the attributes of all cl's parents in the produced vertex.
- **R4:** each association $as \in As$ between two classes $c_1, c_2 \in Cl$ with $reg(c_1) = reg(c_2) = Graph$ is mapped

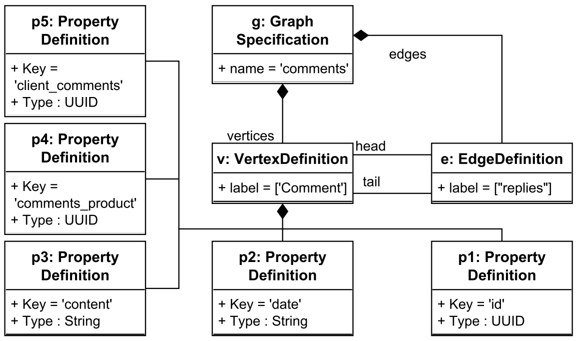

Fig. 8. Mapped GraphDB Instance

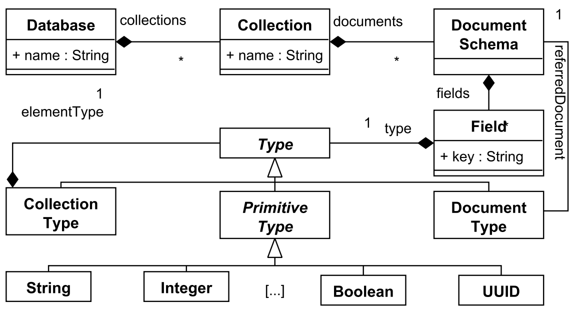

Fig. 9. DocumentDB Metamodel

to an edge definition $e \in E$, where $e.label = as.name$, $e.tail = v_1$, and $e.head = v_2$, where v_1 and v_2 are the *VertexDefinitions* representing c_1 and c_2. Note that $e.tail$ and $e.head$ values are set according to the direction of the association. Aggregation associations are mapped the same way, but their semantic is handled differently at the application level. In order to support inherited associations, *EdgeDefinitions* are also created to represent associations involving the parents of c.

- **R5:** each association $as \in As$ between two classes $c_1, c_2 \in Cl$ with $reg(c_1) = Graph, reg(c_2) \neq Graph$ is translated to a property $p \in P$ where $p.key = as.name$, $p.type = UUID$ and added to the property list of its mapped container v such as $p \in v.properties$.

Figure 8 shows the result of applying this mapping rules on our running example. The created model defines a *Graph-Specification* mapped from the *comments* region, and contains a single *VertexDefinition* `v` mapped from the class *Comment*. This *VertexDefinition* is associated to a single *EdgeDefinition* `e` representing the *replies* association. `V` contains a single label representing its class, and a *PropertyDefinition* `p1` holding its UUID identifier. `V` also contains two *PropertyDefinitions* corresponding to its *content* and *date* attributes, and two additional *PropertyDefinitions* `p4` and `p5` representing references to the *Clients* and *Products* classes, respectively stored in a relational and a document database.

C. Document Database Mapping

1) DocumentDB Metamodel: Figure 9 shows the *DocumentDB metamodel*, our contribution to represent the internal structure of document-oriented datastores. A *Database* is defined by a *name*, and contains a set of *Collections* representing families of *DocumentSchemas*. *DocumentSchemas* are low-level data structures that can be seen as associative arrays containing a set of *Fields*. A *Field* is identified by a *key*, and a *Type* of data it can hold. In our metamodel, *Types* are categorized into three groups: (i) *PrimitiveTypes* representing primitive type values (including the UUID introduced before), (ii) *DocumentTypes* representing *document* references, (iii) *CollectionTypes* representing list of nested elements (containing an *elementType* attribute representing the type of the collection). Note that the *DocumentType* construct is independent of the concrete, platform-specific implementation of document references (e.g. logical link between existing document such as in MongoDB vs. duplication of its content in a nested copy).

2) Class Diagram to DocumentDB Transformation: A Document diagram *DD* is defined as a tuple $DD = (C, D, F)$, where *C* the set of collections, *D* is a set of documents, and *F* the set of documents' fields. In addition, we define the *CollectionType* and *DocumentType* constructs that are derived from the *DocumentDB* metamodel, and are initialized with a single parameter, representing respectively their *elementType* and *referredDocument*. Based on these definitions, the mapping rules are expressed as:

- **R1:** each document region $r = Document, \forall cl(cl, r) \in R$ is mapped to a document database d such as $d.name = r.name$
- **R2:** each class $cl \in Cl, reg(cl) = Document$, $\nexists x(cl, x) \in I$ is mapped to a collection $c \in C$ such as $c.name = cl.name$. Note that this rule produces a collection for each superclass in the model.
- **R3:** each class $cl \in Cl$ (including the ones mapped by R2) are mapped to a document schema $d \in D$ and its containing collection is set as $d.collection = c$, where c represents the mapped collection of the top-level element in cl inheritance hierarchy. As a result, document schemas mapped from classes in the same inheritance hierarchy will be contained in the same collection. This mapping rule also creates a field $f_{id} \in F, f_{id} \in d.fields, f_id.name =' \_id', f_{id}.type = UUID$ representing the unique identifier of the element. Finally, an additional field $f_{classe} \in F, f_{classes} \in d.fields, f_{classes}.key =' \_classes', f_{classes}.type = String$ is also created to store the names of the classes in cl's hierarchy, and is required to filter concrete subclasses from generic super-class collections.
- **R4:** each attribute $a \in (c \cup parents(c)).attributes$ is mapped to a field f, where $f.key = a.name$, $f.type = a.type$, and added to the property list of its mapped container d such as $f \in d.fields$. Multi-valued attribute are mapped similarly, using the *CollectionType* construct to represent multiple values: $f.type =$

$CollectionType(a.type)$.

- **R5:** each association $as \in As$ between two classes $c_1, c_2 \in Cl, reg(c_1) = reg(c_2) = Document$ is mapped to two fields $f_{c1}, f_{c2} \in F$ where $f_{c1} \in d_1.fields, f_{c1}.key = a.name$, $f_{c2} \in d_2.fields, f_{c2}.key = a.name$, $f_{c1}.type = DocumentType(d_2)$, $f_{c2}.type = DocumentType(d_1)$, where d_1 and d_2 are the *Documents* representing c_1 and c_2. This rule creates *Fields* in the documents representing the classes involved in the association, and set their type as a *DocumentType* referring to the other end of the association. In order to support inherited associations, *Fields* are also created to represent associations involving the parents of c. Note that multi-valued ends of the associations are mapped as multi-valued attributes using the *CollectionType* construct: $f_{c1}.type = CollectionType(DocumentType(d_2))$.
- **R6:** each association between two classes from heterogeneous datastores $c_1, c_2 \in Cl, reg(c_1) = Document, reg(c_2) \neq Document$ is mapped to a field $f_{c1} \in F$ where $f_{c1} \in d_1.fields, f_{c1}.key = a.name, f_{c1}.type = UUID$. Multi-valued ends of the associations are mapped similarly to R5. Note that this initial version of our mapping does not consider associations involving more than two classes.
- **R7:** each association class $ac \in Ac$ between classes $cl_1...cl_n$ is mapped to a document d_{ac} such as $d_{ac}.collection = c, c \in C, c.name = ac.name$. As for a regular class, d_{ac} contains the *Fields* corresponding to the attributes $ac.attributes$, and a set of *Fields* $f_{aci} \in F$ where $f_{aci}.key = c_i.name, f_{aci}.type = DocumentType(d_i)$ linking to each document representing a class involved in the association. In addition, an identifier *Field* $f_{id} \in F$ is also produced such as $f_{id} \in f_{ac}.fields, f_{id}.name = '\_id', f_{id}.type = UUID$ and added to the contained fields. Multi-valued ends are mapped as multi-valued attributes using the *CollectionType* construct: $f_{ci}.type = CollectionType(DocumentType(d_{ac}))$.

To better illustrate the different mapping rules, we present in Listing 1 the output of the *UmlToDocumentDB* translation when applied on the running example. For clarity, the resulting *DocumentDB* model is shown using a JSON-like concrete syntax derived from the metamodel (abstract syntax) presented in Figure 9. A *Database* instance named *business* contains three collections representing the classes of the input model: *order*, *product*, and the association class *orderLine*. The *order* collection defines a single document schema *orderDoc* which contains a set of *Fields* mapping *Order* class attributes in the conceptual schema. *orderDoc* also includes a multi-valued document reference to *orderLineDoc*, as a result of the mapping of the relationship between Order and the OrderLine association class, and an external reference to *client* (materialized as a UUID), as part of the mapping between Order and the Client class, located in a different region in the conceptual schema.

```
Database "business" {
  Collection "order" {
    DocumentSchema orderDoc {
      _id : UUID
      _classes : [String]
      reference  : String
      shipmentDate : Date
      deliveryDate : Date
      paid : Boolean
      client : UUID
      orderLine : [orderLineDoc]
  } }
  Collection : "product" {
    DocumentSchema productDoc {
      _id : UUID
      _classes : [String]
      name : String
      price : Integer
      description : String
      orderLine : [orderLineDoc]
  } }
  Collection : "orderLine" {
    DocumentSchema orderLineDoc {
      _id : UUID
      _classes : [String]
      quantity : Integer
      productPrice : Integer
      orderLine_order : orderDoc
      orderLine_product : productDoc
} } }
```

Listing 1. Mapped DocumentDB Model (Textual Syntax)

```
context Product inv validPrice: self.price ≥ 0
context Comment inv validComment: self.date > self.
     repliesTo.date
context Order inv validOrder: if self.paid then self.
     orderLine→forAll(o | o.quantity > 0) else false endif
context Client inv maxUnpaidOrders: self.orders→select(o |
      not o.paid)→size()< 3
```

Listing 2. Textual Constraints

V. Mapping OCL Expressions

To complete the PIM-to-PSM transformation process, the OCL constraints attached to the input conceptual schema are translated to queries expressed in the query language/s available in the data stores where the model elements affected by the query will be mapped.

In this section, we first discuss the translation of OCL queries local to a single region and, later, how the process deals with global OCL constraints, i.e. constraints referencing model elements in different regions. Note that the generated queries can be manually integrated in an existing database infrastructure, or wrapped in constraint checking methods provided by the runtime component presented in Section VI.

To illustrate this section, we will use the four OCL constraints of Listing 2 defined on top of our running example (Figure 4). The first one checks that the *price* of a *Product* is always positive, the second one checks that the date of a comment is always greater than its parent one, the third one verifies that once an *Order* has been *paid* all its *orderLines* have a positive *quantity* value, and the last one ensures that a *Client* has less than three unpaid *Orders*. Note that constraints 1-3 target a single region, while the last one references elements from the relational and document datastores.

TABLE I
OCL TO SQL MAPPING

OCL expression	SQL Fragment
Type	"Type.name"
C.allInstances()	select * from C
o.collect(attribute)	select attribute from C where id in o.id
o.collect(reference) (single valued)	select reference from C where id in o.id
o.collect(reference) (multi valued)	select * from reference where reference_C_id in o.id
size()	count()
$col_1 \rightarrow$union(col_2)	col_1 union col_2
$col_1 \rightarrow$including(o)	col_1 union select * from o.class where id = o.id
$col_1 \rightarrow$excluding(o)	col_1 except select * from o.class where id = o.id
col$\rightarrow$select(condition)	select * from o.class where condition
col$\rightarrow$reject(condition)	select * from o.class where not(condition)
=, >, >=, <, <=, <>	==, >, >=, <, <=, <>
+, -, /, %, *	+, -, /, %, *
and, or, not	and, or, not
literals	*literals*

```
select * from product where price < 0
```

Listing 3. Generated SQL Query from the validPrice Constraint

Currently, UMLto[No]SQL supports three simple mappings: *OCLToGraphDB* [9] that complements the *UmlToGraphDB* transformation by generating graph queries, *OCLToSQL* that is adapted from existing work on UML/OCL translation to relational models [3], [11], and *OCLToMongoDB* that maps OCL constructs to the MongoDB query language [7]. Below we detail this last mapping, which is a new contribution of this paper.

A. Relational Query Mapping

SQL [16] is the obvious target for our translation process on relational databases. Given the (mostly) universal support for this standard language, our generated SQL expressions will be useful no matter what particular RDBMS vendor is used.

Table I shows an excerpt of the OCL to SQL mapping used in our translation process [3]. This mapping is adapted from the one presented by Demuth et al. [11].

Specifically, attribute and single-valued association navigations are mapped to projections of the corresponding column in the table containing instances of the input object. Instance filtering is performed based on the *id* field defined in our relational metamodel. Element selections are mapped into *select* statements, with a *where* clause containing the translation of its condition. Collection operators (such as *union*), mathematical operations, logical operators and literals are direclty translated to their SQL equivalent.

The created fragments are combined into a query that retrieves all the records which do not satisfy the constraint, a common pattern when evaluating OCL constraints over models [11]. As an example, Listing 3 shows the result of applying our OCL to SQL translation on the first constraint of our running example.

[3]We focus on the mappings applicable to the running example for brevity.

TABLE II
OCL TO GREMLIN MAPPING

OCL expression	Gremlin step
Type	"Type.name"
C.allInstances()	g.V().hasLabel("C.name")
collect(attribute)	property(attribute)
collect(reference)	outE('reference').inV
oclIsTypeOf(C)	o.hasLabel("C.name")
$col_1 \rightarrow$union(col_2)	col_1.fill(var_1); col_2.fill(var_2); union(var_1, var_2);
including(object)	gather{it << object;}.scatter;
excluding(object)	except([object]);
size()	count()
isEmpty()	toList().isEmpty()
select(condition)	c.filter{condition}
reject(condition)	c.filter{!(condition)}
exists(expression)	filter{condition}.hasNext()
=, >, >=, <, <=, <>	==, >, >=, <, <=, !=
+, -, /, %, *	+, -, /, %, *
and,or,not	&&,||,!
variable	variable
literals	*literals*

B. Graph Query Mapping

1) Gremlin Query Language: Gremlin is a Groovy domain-specific language built over *Pipes*, a lazy data-flow framework on top of *Blueprints*. We have chosen Gremlin as the target query language for our graph query translation due to its adoption in several graph databases.

Gremlin is based on the concept of process graphs. A process graph is composed of vertices representing computational units and communication edges which can be combined to create a complex processing flow. In the Gremlin terminology, these complex processing flows are called *traversals*, and are composed of chains of simple computational units named *steps*. Gremlin define four types of steps: *transform steps* that map inputs of a given type to output of another one, *filter steps*, selecting or rejecting input elements according to a given condition, *branch steps*, which split the computation into several parallel sub-traversals, and *side-effect steps* that perform operations like edge or vertex creation, property update, or variable definition or assignments.

In addition, the step interface provides a set of built-in methods to access meta information: number of objects in a step, output existence, or first element in a step. These methods can be called inside a traversal to control its execution or check conditions on particular elements in a step.

2) OCL to Gremlin Transformation: Table II presents the mappings between OCL expressions and Gremlin concepts. Supported OCL expressions are divided into four categories based on Gremlin step types: transformations, collection operations, iterators, and general expressions. As before, we only present a subset of the OCL expressions we support[4].

These mappings are systematically applied on the input OCL expression, following a postorder traversal of the OCL syntax tree. Listing 4 shows the Gremlin query generated from the third OCL constraint of our running example. The v

[4]A complete version of this mapping is available in our previous work [10]

variable represents the vertex that is being currently checked, and the following steps are created using the mapping.

C. Document Query Mapping

1) MongoDB Query Language: We have chosen the MongoDB query language as a representative of the family of query languages for document databases. A similar mapping could be implemented for other document query languages. Together with the DocumentDB metamodel presented in the previous section, this OCL transformation enables a complete translation of UML/OCL schemas to document databases. As we said above, this transformation complements other available transformations from UML to other relational and NoSQL backends, including mixed solutions.

The MongoDB Query Language is a Javascript-based language that defines additional constructs to retrieve documents within a collection, filter them, or retrieve fields given a specific condition. This language is embedded in the MongoDB shell, and can be sent for computation to a MongoDB server.

2) OCL to MongoDB Query Language Transformation: Table III shows the OCL to MongoQL mapping (again, focusing on a number of the most relevant OCL operations; many others can be easily reexpressed in terms the ones listed here [5]).

In the following, we explain these mappings using the example constraints introduced before. Note that we apply a similar translation process as we did for the SQL mapping, i.e. the generated queries return the instances violating the constraints.

According to this, Listing 5 shows how the first constraint is mapped to a `db.product.find()` method with a *where* condition that traverses all documents in the product collection and returns those with a negative price (if any). Second constraint includes an *if-then* construct to first filter out unpaid orders. For the paid ones, the translation proceeds with a document lookup (mapped to a nested collection lookup in MongoDB) to retrieve all the order lines of the order being processed searching for at least one that violates the *forAll* condition in the original OCL expression. If found, the order is returned as part of the query result. Note that the *find* operation is one of the cornerstones of our mapping since it is expressive enough to be the mapping target of multiple OCL operations.

In addition, we reuse the Javascript native support of the query language to map general expressions such as arithmetic and logical operators, comparisons, literals, and variables. We also define the *union*, *intersection*, and *set subtraction* operations, that can be called as regular functions. Finally, our mapping relies on the methods *push* and *splice* of Javascript arrays to translate *including* and *excluding* operations.

D. Cross-Datastore Query Mapping

Cross-datastore queries are OCL expressions that reference elements in separate regions. Typically, they include a navigation operation from an element in a region to an element in a neighbour region. Our approach handles such queries by: 1) translating datastore-specific fragments of the input expression into native database queries, and 2) providing a composition mechanism to complete their overall evaluation.

```
v.property('date')>v.outE('repliesTo').inV.property('date')
```

Listing 4. Generated Gremlin Query from the validComment Constraint

```
db.product.find({$where: "this.price ≤ 0"}) // validPrice
db.order.find({_id: {$where:
  "if(this.paid) {
    !(db.orderLine.find({_id: {$in : db.order.find({$where:
         "_id == this._id"}, {orderLine : 1})}, $where: "
        this.quantity ≤ 0"}).hasNext());
  } else {false}"}}); // validOrder
```

Listing 5. Generated MongoDB Queries from the Running Example

```
Context Client inv maxUnpaidOrders:
// clients region (Relational)
let orders : Sequence(Order) =
  self.orders
in
// business region (Document)
  orders→select(o | not o.paid)→size()<3
```

Listing 6. Rewritten OCL Constraint Example

To do so, we first perform a simple rewriting of the OCL expression that encapsulates each datastore-specific operation sequence in an intermediate variable. This initial step decouples datastore-specific OCL fragments, and allows to easily translate the value of each variable using our mappings. Listing 6 shows the result of this initial processing for the third query of our running example: a variable *orders* is created to store the results of the `self.orders` navigation (from the relational region), and the `in` clause reuses the created variable to compute the `select` operation (in the document region).

This refactored OCL expression is then translated into an instance of our *Cross-Datastore Query Metamodel* (Figure 10) that provides primitives to compose native database queries. Specifically, a cross-datastore script (*CDScript*) is composed of an ordered collection of *instructions* representing *Function* calls. A *Function* takes a set of *Variables* as *parameters* and returns a *result* which is also stored in a *Variable*. This first version of our metamodel supports two kind of *Functions*: *NativeQueries* that are produced by the native translations presented above, and *JoinFunctions* that handle the computation of cross-region associations. Generate *NativeQueries* take has parameter the result of the previously executed *JoinFunction*, allowing to push most of the data processing to the datastores. Note that the concrete implementation of *JoinFunctions* is discussed in the next section.

Instances of the Cross-Datastore Query Metamodel are generated by mapping each `let source` expression to a *NativeQuery*, and each `in` clause to a *JoinFunction* adapting the previous query's results, followed up by a the *NativeQuery* generated from its inner expression. Figure 11 shows the result of this translation for the *maxUnpaidOrders* constraint of our running example, which contains an initial *SQLQuery* computing the *orders* navigation, a *JoinFunction* that adapts the *Orders* returned by the relational database to documents, and a final *MongoQuery* that computes the *Order* selection.

TABLE III
OCL TO MONGODB

OCL expression	MongoDB Query Fragment
Cl.allInstances()	db.C_{cl}.find()
o.$attr_a$	db.$C_{type(o)}$.find({$where : "_id == o"},{$attr_a$: 1})
o.ref_a	db.$C_{type(refa)}$.find({_id: {$in: db.$C_{type(o)}$.find({$where : "_id == o"},{ref_a: 1})}})
o.oclIsTypeOf(Cl)	db.C_{cl}.contains(o)
col_1->union(col_2)	union(col_1, col_2)
col_1->intersection(col_2)	intersection(col_1, col_2)
col_1 - (col_2) (set subtraction)	subtract(col_1, col_2)
col->including(o)	col.push(o)
col->excluding(o)	col.splice(col.indexOf(o), 1)
col->select(condition)	db.C.find({_id: {$in: col}, $where : "condition"})
col->reject(condition)	db.C.find({_id: {$in: col}, $where : "!(condition)"})
col->exists(expression)	db.C.find({_id: {$in: col}, $where : "condition"}).hasNext()
col->forAll(expression)	!(db.C.find({_id: {$in: col}, $where : "!(condition)"}).hasNext())
col->size()	C.length
col->isEmpty()	C.length == 0
if(c) then e_1 else e_2 endif	if(c) { e_1} else {e_2}
=, >, >=, <, <=, <>	==, >, >=, <, <=, !=
+, -, /, %, *	+, -, /, %, *
and, or, not	&&, ||, !
variable	variable

Legend — Cl: class; o: object; col: *object* collection; C_a: *database* collection containing objects of type *a*; the *:1* suffix is a boolean flag to indicate the expression should return only the indicated field and not the whole document.

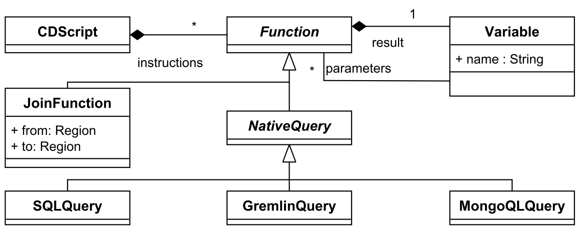

Fig. 10. Cross-Datastore Query Metamodel

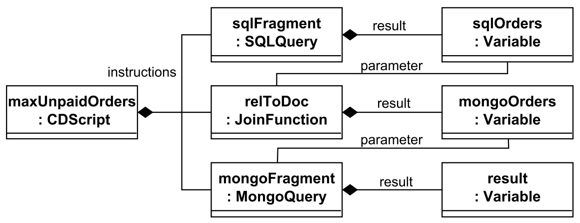

Fig. 11. Cross-Datastore Query Instance

VI. CODE GENERATION

As a final step in UMLto[No]SQL, a code generation process takes as input the previous PSMs and query models and generates the software artifacts to guarantee the proper implementation of such models in the chosen data stores.

The concrete artifacts to generate depends on the type of storage solution. For instance, for relational databases, most of the code generation focuses on creating the SQL DDL scripts to setup the database. Instead, for NoSQL databases (which are mostly schemaless), a large part of the generation process is devoted to producing application code that integrates and enforces the designed structures and constraints. To ease this process, we provide a generic *Modeling Layer* that defines primitives to persist, query, and update data in NoSQL datastores. The generated application code is then plugged into this generic infrastructure, and extends it to provide additional operations derived from the conceptual schema (e.g. getting all the *clients* stored in the system).

Figure 12 shows the architecture of our generic *Modeling Layer*. The *Middleware* class is the central component that allows to create, search, and delete *Beans* representing the model elements at the PIM level. It contains a set of *Datastores* that are responsible for the raw database initialization and accesses. Currently, our framework embeds three *Datastore* subclasses, that allow to access MongoDB, PostgreSQL, and Blueprints. The *Middleware* also embeds a set of *QueryProcessors*, that are responsible of the constraint computation. *NativeProcessors* support the execution of native queries over their associated *Datastore*, and *CDProcessor* handles the computation of cross-datastore queries. The *CDProcessor* relies on a set of *NativeProcessors* that are used to compute the native fragment of the composite query, and uses the *join* method provided by the *Datastores* to compute the join function introduced in Section V. This method retrieves the stored records matching the provided *UUIDs* and *type*, that can be provided as input of the next composite query computation defined in the *CDQuery*. Note that these functions do not return *Bean* instances, but native objects that can be handled by the *Datastore* to compute a query.

The UMLto[No]SQL code generator creates additional classes that extend the ones shown in Figure 12 (light-grey nodes). Individual *Beans* are created, wrapping the classes defined in the conceptual schema, allowing to natively manipulate them. In our example, this process generates the

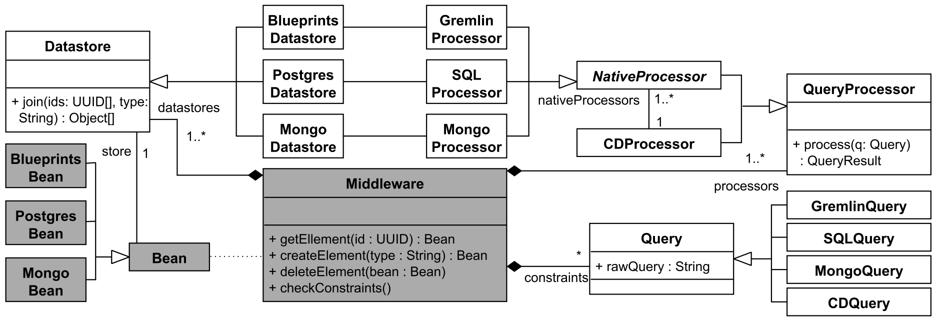

Fig. 12. UMLto[No]SQL Modeling Layer Infrastructure

Client, *Order*, *Product*, and *OrderLine* beans, and creates a set of methods to access and update their attributes and related associations by delegating the operation transparently on the corresponding datastore. Finally, a specific implementation of the *Middleware* class containing the generated constraint *Queries*, and initialized with the *Datastores* and *QueryProcessors* associated to the conceptual schema is also generated.

We would like to remark that an important benefit of this middleware is that it enables users to access the data using the same concepts and types they employed when modeling the domain, regardless of how those model elements have been refined and transformed to implement them in the data store. Still, our approach can be used to simply generate standalone database artifacts, that could then be integrated into separate solutions.

```
public class Order extends MongoBean {
  public Order(ObjectId id, MongoDatastore mongoDatastore){
    super(id, mongoDatastore);  }
  public Date getShipmentDate() {
    Long timestamp = getValue("shipmentDate");
    return new Date(timestamp);  }
  public void setShipmentDate(Date newShipmentDate) {
    Long timestamp = newShipmentDate.getTime();
    updateField("shipmentDate", timestamp);  }
  // Cross-datastore reference
  public Client getClient() throws ConsistencyException {
    ObjectId clientId = getValue("client");
    return DemoMiddleware.getInstance().getClient(clientId.
         toString());  }
  public void setClient(Client newClient) {
    if(isNull(newClient))
      throw new ConsistencyException("Cannot␣set␣null␣
           Client,␣the␣association␣cardinality␣is␣1");
    String clientId = newClient.getId();
    ObjectId oId = new ObjectId(clientId);
    updateField("client", oId);  }
}
```

Listing 7. Generated Order Class

As an example, the Listing 7 shows an excerpt of the code generated for the class *Order*. The generated bean extends the *MongoBean* class, that provides generic methods such as *getValue* and *updateField*, and is responsible of the connection to the underlying *MongoDatastore*. Getters and setters are generated for all the attributes and associations. The *getClient* method shows how cross-datastore associations are handled: the *UUID* stored in the the *Order* document is deserialized into a String, that is used to retrieve the *Client* instance thanks to the *Middleware* singleton class (which delegate the search to the relational database managing *Client* instances). A similar process is followed to implement the constraints. See the project repository for a full implementation of the example.

VII. Related Work

UMLto[No]SQL can be related to several works focusing on the mapping of conceptual schemas to databases. We propose here a classification of relevant approaches into two families: generative approaches that translate conceptual schemas to data storage solutions, and abstraction layers built on top of existing backends that aim to unify them under a common model/API that could be used to simplify the mapping.

The mapping of conceptual schemas to relational databases is a well-studied field of research [20]. A few works also cover (OCL) constraints. For example, Demuth and Hussman [11] provide a code generator from UML/OCL to SQL [12]. Similarly, Brambilla et al. [3] propose a methodology to implement integrity constraints into relational databases recommending alternative implementations based on performance parameters. Fewer works address non-relational databases. Zhao et al. [29] propose an approach to model MongoDB databases starting from a relational model. Li et al.focus on the transformation from UML class diagrams into HBase [19] while NoSQL Schema Evaluator [21] focuses on generating column family definitions and query implementation plans from a conceptual schema optimized for a given workload. Nevertheless, none of these approaches support multiple data stores nor provide support for the constraint mapping part.

On the other hand, some approaches try to provide some kind of modeling language or abstraction to simplify the interaction with NoSQL databases. Bugiotti et al. [4] propose a design methodology for NoSQL databases based on NoAM, an abstract data model that aims to represent NoSQL systems in a system-independent way. NoAM models can be implemented in several NoSQL databases. Federated data management and querying approaches can also be related to our work [13]. For instance, CloudMdsQL [17] is a SQL-like query language capable of querying multiple data stores within a single query. To do so, the language extends the standard SQL syntax with additional constructs allowing to call native queries that target a specific datastore. Apache Drill [15] is a similar framework that provides an extended SQL syntax to query multiple data sources including relational databases, Json documents, or document databases. These approaches could be potentially useful as intermediate representations in our own mapping process as a way to enlarge the number of stores that we could support but we believe our UML-based approach offers a lower barrier of entry since it allows designers to model their schemas using well-known notations (and tools, most likely already part of their standard modeling tool chain) instead of requiring them to learn new languages and adhoc solutions from the very beginning.

VIII. Conclusion and Future Work

In this paper we have presented UMLto[No]SQL, a MDA-based framework to partition and map conceptual schemas to several data storage solutions. Our approach combines model mapping techniques with a set of rules to translate OCL constraints into various native query languages, including an ad-hoc mechanism to compute multi-platform queries. Our code generator hides all internal details of the chosen data stores, allowing designers to keep a unified view of the model regardless the actual data storage.

As future work, we plan to evaluate our approach through real-world use cases, and benefit from the modular architecture of our framework to add more data storage options. We also want to improve the flexibility of our approach by proposing PSM-level refactoring operations to let designers tune the mappings according to specific needs (e.g. performance requirements, availability, data replication, data locality, etc.). In addition, we plan to benchmark the performance of the produced queries, especially cross-datastore ones. In this sense, we also envision to reuse advanced query composition techniques such as CloudMdsQL [17] or Apache Drill [15].

Another ongoing work is the integration of automatic schema partitioning techniques according to an expected/measured workload. This optimization problem has been heavily studied in the context of relational database physical schema design [1], [14], and can be adapted to define regions on the conceptual schema. Finally, we would like to work on the reverse direction, i.e. the automatic extraction of conceptual schemas from an existing set of datastores. This is a complex problem even when targeting a single NoSQL data store (e.g. see works such as [26] or [8]) that becomes much harder to tackle when considering the relationships between data across different data stores.

Acknowledgement

This work was supported by the *Programa Estatal de I+D+i Orientada a los Retos de la Sociedad* Spanish program (Ref. TIN2016-75944-R); and the ECSEL Joint Undertaking under grant agreement No. 737494 – MegaM@Rt2. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation program and from Sweden, France, Spain, Italy, Finland & Czech Republic.

References

[1] S. Agrawal, V. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 359–370. ACM, 2004.

[2] G. Blair, N. Bencomo, and R. France. Models@run.time. *Computer*, 42(10), 2009.

[3] M. Brambilla and J. Cabot. Constraint Tuning and Management for Web Applications. In *Proc. of the 6th ICWE Conference*, pp. 345–352, 2006.

[4] F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone. Database Design for NoSQL Systems. In *Proc. of the 33rd ER Conference*, pp. 223–231. Springer, 2014.

[5] J. Cabot and E. Teniente. Transformation Techniques for OCL Constraints. *SCP*, 68(3):179 – 195, 2007.

[6] R. Cattell. Scalable SQL and NoSQL Data Stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[7] K. Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media, Inc., 2013.

[8] I. Comyn-Wattiau and J. Akoka. Model Driven Reverse Engineering of NoSQL Property Graph Databases: The Case of Neo4j. In *Proc. of Big Data 2017*, pp. 453–458. IEEE, 2017.

[9] G. Daniel, G. Sunyé, and J. Cabot. UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases. In *Proc. of the 35th ER Conference*, pp. 430–444. Springer, 2016.

[10] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Mogwaï: a framework to handle complex queries on large models. In *Proc. of the 10th RCIS Conference*, pp. 1–12. IEEE, 2016.

[11] B. Demuth and H. Hussmann. Using UML/OCL Constraints for Relational Database Design. In *Proc. of the 2nd UML Conference*, pp. 598–613. Springer, 1999.

[12] B. Demuth, H. Hußmann, and S. Loecher. OCL as a Specification Language for Business Rules in Database Applications. In *Proc. of the 4th UML Conference*, pp. 104–117. Springer, 2001.

[13] Olaf Görlitz and Steffen Staab. Federated data management and query optimization for linked open data. In *New Directions in Web Data Management 1*, pp. 109–137. Springer, 2011.

[14] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: a Main Memory Hybrid Storage Engine. *Proc. of the VLDB Endowment*, 4(2):105–116, 2010.

[15] M. Hausenblas and J. Nadeau. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big Data*, 1(2):100–104, 2013.

[16] International Organization for Standardization. ISO/IEC 9075-1:2016, 2019. URL: https://www.iso.org/standard/63555.html.

[17] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. Pereira. CloudMdsQL: Querying Heterogeneous Cloud Data Stores with a Common Language. *DAPD*, 34(4):463–503, 2016.

[18] François Lagarde, Huáscar Espinoza, François Terrier, and Sébastien Gérard. Improving uml profile design practices by leveraging conceptual domain models. In *Proc. of the 22nd ASE Conference*, ASE '07, pp. 445–448, New York, NY, USA, 2007. ACM.

[19] Y. Li, P. Gu, and C. Zhang. Transforming UML Class Diagrams into HBase Based on Meta-Model. In *Proc. of the 4th ISEEE Conference*, volume 2, pp. 720–724. IEEE, 2014.

[20] E. Marcos, B. Vela, and J.M. Cavero. A Methodological Approach for Object-Relational Database Design Using UML. *SoSyM*, 2(1):59–72, 2003.

[21] M.J. Mior, K. Salem, A. Aboulnaga, and R. Liu. NoSE: Schema Design for NoSQL Applications. *IEEE TKDE*, 29(10):2275–2289, 2017.

[22] A. Moniruzzaman and S.A. Hossain. NoSQL Database: New Era of Databases for Big Data Analytics - Classification, Characteristics and Comparison. *IJDTA*, 6(4):1–14, 2013.

[23] OMG. MDA Specifications, 2018. URL: http://www.omg.org/mda/specs.htm.

[24] OMG. About the Object Constraint Language Specification Version 2.4, 2019. URL: https://www.omg.org/spec/OCL/2.4/.

[25] OMG. About the Unified Modeling Language Specification Version 2.5.1, 2019. URL: https://www.omg.org/spec/UML/2.5.1/.

[26] D. Ruiz, S. Morales, and J. Molina. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Proc. of the 34th ER Conference*, pp. 467–480. Springer, 2015.

[27] B. Selic. A systematic approach to domain-specific language design using uml. In *Proc.of the 10th ISORC Symposium*, volume 00, pp. 2–9, 05 2007.

[28] Tinkerpop. Tinkerpop Website, 2018. URL: www.tinkerpop.com.

[29] G. Zhao, W. Huang, S. Liang, and Y. Tang. Modeling MongoDB with Relational Model. In *Proc. of the 4th EIDWT Conference*, pp. 115–121. IEEE, 2013.