TYPHON
POLYGLOT AND HYBRID PERSISTENCE ARCHITECTURES FOR BIG DATA ANALYTICS

Project Number 780251

D6.3 Hybrid Polystore Data Migration Tools

Version 1.0
29 June 2019
Final

Public Distribution

University of Namur

Project Partners: **Alpha Bank**, **ATB**, **Centrum Wiskunde & Informatica**, **CLMS**, **Edge Hill University**, **GMV**, **Nea Odos**, **The Open Group**, **University of L'Aquila**, **University of Namur**, **University of York**, **Volkswagen**

Project Partner Contact Information

Alpha Bank Vasilis Kapordelis 40 Stadiou Street 102 52 Athens Greece Tel: +30 210 517 5974 E-mail: vasileios.kapordelis@alpha.gr	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de
Centrum Wiskunde & Informatica Tijs van der Storm Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 9333 E-mail: storm@cwi.nl	**CLMS** Antonis Mygiakis Mavrommataion 39 104 34 Athens Greece Tel: +30 210 619 9058 E-mail: a.mygiakis@clmsuk.com
Edge Hill University Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk	**GMV Aerospace and Defence** Almudena Sánchez González Calle Isaac Newton 11 28760 Tres Cantos Spain Tel: +34 91 807 2100 E-mail: asanchez@gmv.com
Nea Odos Charalampos Daskalakis Themistocleous 87 106 83 Athens Greece Tel: +30 210 344 7300 E-mail: cdaskalakis@neaodos.gr	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L'Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**University of Namur** Anthony Cleve Rue de Bruxelles 61 5000 Namur Belgium Tel: +32 8 172 4963 E-mail: anthony.cleve@unamur.be
University of York Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk	**Volkswagen** Behrang Monajemi Berliner Ring 2 38440 Wolfsburg Germany Tel: +49 5361 9-994313 E-mail: behrang.monajemi@volkswagen.de

Document Control

Version	Status	Date
0.1	Document outline	21 March 2019
0.5	First draft	12 June 2019
0.8	Full draft	21 June 2019
1.0	Final version after partner review	29 June 2019

Table of Contents

Executive Summary

In the context of its Work Package 6, the TYPHON project aims to develop a methodology and technical infrastructure to support the graceful evolution of hybrid polystores, where multiple, possibly overlapping NoSQL and SQL databases may co-evolve in a consistent manner.

The proposed methodology should cover four main aspects: (1) polystore schema evolution: Allowing the TyphonML polystore schema to evolve over time in response to changes in terms of data requirements; (2) Polystore data migration: Allowing data to be migrated from one (version of) a datastore to another (version) of a datastore; (3) polystore query migration: Allowing to automatically support the adaptation of existing TyphonQL queries to an evolving polystore schema; (4) continuous polystore evolution: exploiting the polystore query events captured by the monitoring mechanisms developed in WP5 in order to recommend possible polystore schema reconfigurations (be they intra-paradigm or inter-paradigm).

This deliverable covers the design and implementation of tool support for the two first aspects, namely the evolution of the TyphonML polystore schema and the associated migration of data from one polystore schema version to another.

1 Introduction

According to the Work Package 6, the TYPHON project aims at developing a methodology and technical infrastructure of hybrid polystore Data Migration tools in order to ensure an automated support of cross-database and cross-paradigm data migration. It takes into account the evolution of hybrid polystores, where multiple, NoSQL and SQL databases may co-evolve in a consistent manner.

The objective of this Work Package includes the development of methods and tools for:

1. migrating data across database paradigms and across database platforms (relational, NoSQL);
2. evolving the data organization and distribution in hybrid persistence architectures;
3. monitoring the actual use of hybrid data stores in order to inform the continuous evolution process.

In order to reach this purpose, the proposed approaches for addressing these different processes aim to be *as transparent as possible* for the client applications.

The TYPHON polystore evolution tools will support four main aspects:

- Polystore schema evolution: Allowing the TyphonML polystore schema to evolve over time in response to changes in terms of data requirements.
- Polystore data migration: Allowing data to be migrated from one version of a datastore to another version of a datastore.
- Polystore query migration: Allowing au automated support of the adaptation of existing TyphonQL queries to an evolving polystore schema.
- Continuous polystore evolution: exploiting the polystore query events captured by the monitoring mechanisms developed in WP5 in order to recommend possible polystore schema reconfigurations (be they intra-paradigm or inter-paradigm).

1.1 Purpose of the deliverable

This document presents the work that has been done with respect to task 6.3 of Work Package 6, described as follows in the TYPHON Description of Work:

Task 6.3: The focus of this task is the development of methods and tools for cross-database and cross-paradigm data migration. This includes the automatically supported migration of data from one database schema to another (cross-database migration), but also the migration of relational database fragments to NoSQL data stores, and conversely (cross-paradigm migration).

1.2 Relationship to other TYPHON deliverables

In deliverable D6.1, we identified and compared existing approaches, techniques and tools to database schema evolution and data migration. This literature review was an important source of inspiration when designing the TYPHON schema evolution and data migration methodology presented deliverable D6.2.

The present deliverable present the schema evolution and migration tools, that implements the methodology presented in deliverable D6.2. Those tools automate, in particular, the Schema Modification Operators (SMOs) that are fully specified in deliverable D6.2.

The TYPHON schema evolution and data migration approach is strongly related to the TyphonML data modeling language and its model editor (deliverables D2.1, D2.3 and D2.4) and to the TyphonQL DDL/DML query language (deliverables D4.2 and D4.3).

1.3 Contributors

The main contributor of this deliverable is University of Namur. All project partners contributed to this deliverable, by providing us with input and feedback on earlier versions of the schema evolution and data migration tools, during the Typhon project meetings in Wolfsburg (7-8 March., 2019) and in Athens (6-7 June., 2019).

1.4 Structure of the deliverable

The remainder of this deliverable is structured as follows:

- Section 2 briefly introduces the main concepts needed to present our TYPHON polystore schema evolution and data migratin architecture;
- Section 3 presents the global architecture of the schema evolution and data migration tools, and its relationships with other TYPHON components;
- Section 4 describes the design of the schema evolution and data migration tools;
- Section 5 elaborates on the implementation aspects of those tools;
- Section 6 illustrates a data migration operator at the level of a complete TyphonML entity, based on a concrete example;
- Section 7 summarizes the deliverable and anticipates the next steps of Work Package 6.

2 Preliminaries

In this section, we briefly introduce the main concepts needed to present our TYPHON polystore schema evolution architecture. We first describe TyphonML, the modeling language used to represent the data structures of the TYPHON polystore in a platform-independent manner. Then, we describe TyphonQL, the query language used to manipulate polystore data structures and data instances.

2.1 TyphonML

The TyphonML is the language used to specify a TYPHON polystore.

Following deliverable D2.1 we can identify two different levels of abstraction when designing the polystore data structure by means of a TyphonML model. The first level is the *conceptual* level. This level details the *conceptual entities* and their attributes, those can be of type *dataTypes* which can be either primitive or complex.

The second level relates to the *databases* types, referred to as *logical* level in this document. It represents the different platform-specific databases that will actually store the data instances of the conceptual entities.

Figure 1 shows the meta classes of the TyphonML modeling language taken from deliverable D2.3. A TyphonML schema evolution consists in modifying instances of one of these meta classes or of their attributes. Each type of modification corresponds to an operator such as *add, remove, merge, update*, etc. Each one of these operators, modifying a specific object of a TyphonML schema potentially has impact on the data structure or on the data instances of the polystore. The schema evolution and migration tool is in charge of propagating those evolution impacts.

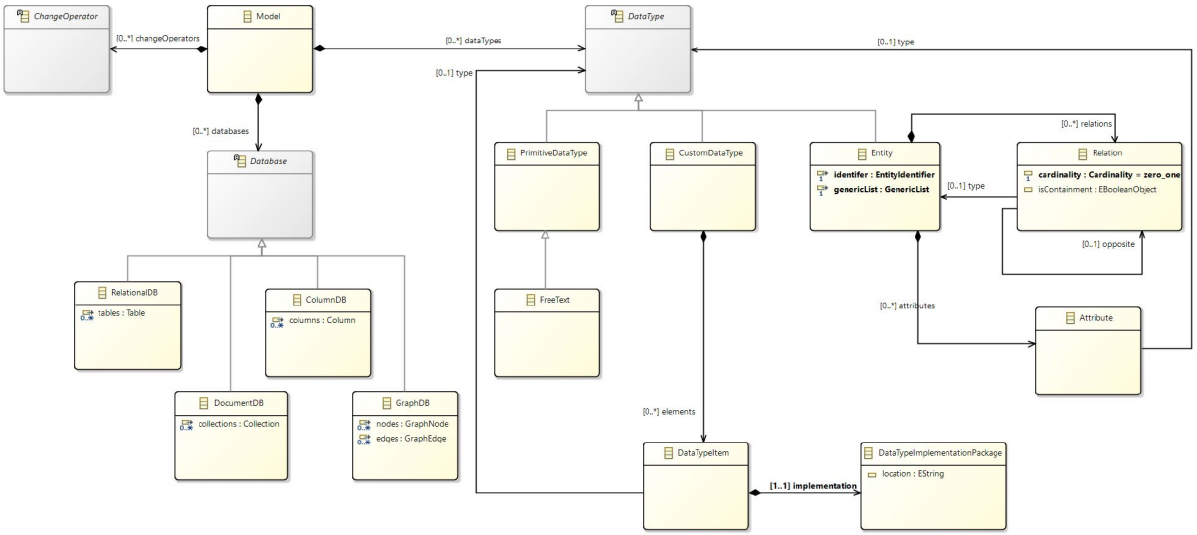

Figure 1: Main Metaclasses of TyphonML language taken from deliverable D2.3

Change Operators

The evolution operations are modeled using *Change Operator* meta class. Each evolution operation is a subtype of this meta class. Instances of these operators can be created by the user in the TyphonML editor using

the evolution mode. The corresponding change operator instances are then saved in the model. When the model is sent to the evolution and migration tool it reads the model and extracts the corresponding operator in order to execute them. Figure 2 shows the meta classes of the available evolution operation available on an Entity TyphonML object.

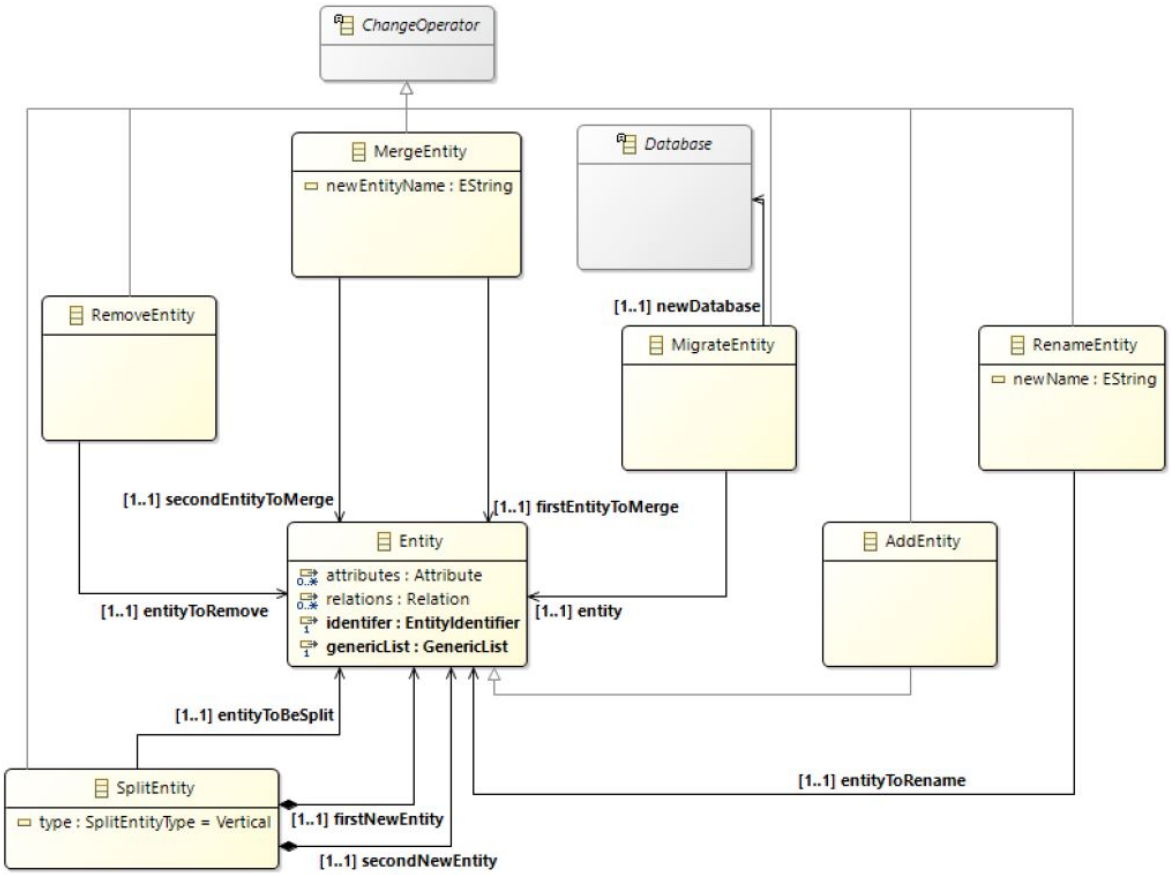

Figure 2: Metaclasses of Entity Change Operators from deliverable D2.3

2.2 TyphonQL

TyphonQL, the TYPHON query language is a unique language that allows the manipulation of data and data structures of underlying databases of a TYPHON polystore. This generic language manipulates TyphonML objects and compiles the query into native, platform-specific queries on the actual databases. The TyphonQL language will be used by the schema evolution and migration tool in order to effectively implement the data and structure changes implied by the evolution operators performed on top of the TyphonML schema.

2.2.1 Data Manipulation Language

In order to perform the read and write operations as well as performing the related data changes required by schema modification operators, a platform-independent Data Manipulation Language (DML) will be offered by

TyphonQL. This language will provide read, write, update, and delete operations on TyphonML entities. The execution of these operations will be possible even when the underlying data are stored in different database paradigms supported by TYPHON (i.e., relational, document-based, key-value stores, graph databases, etc.).

2.2.2 Data Definition Language

In order to perform the data structure changes propagating TyphonML schema modifications, a Data Definition Language (DDL) will also be provided by the TyphonQL language. Create, update, and delete operations will be provided at the level of most TyphonML conceptual objects (entities, attributes, identifiers and indexes).

Note that invoking this TyphonQL DDL does not necessarily imply structural changes applied to the underlying databases. In case of relational and column-oriented databases, propagating a TyphonML schema modification (e.g., adding an attribute to a TyphonML entity) will typically be propagated as data structure changes at the physical database level (e.g., adding a column to the corresponding relational table). However when using schema-less NoSQL databases (e.g., MongoDB), a TyphonML schema change may imply data changes (e.g., updating the *value* of an attribute) instead of data structure changes. In some specific cases, the TyphonML schema changes may have no impact at all on the underlying database structures and contents (e.g Adding an attribute to an entity relying on a document database).

As an example, let us consider the renaming of an TyphonML attribute. If the corresponding TyphonML entity is mapped to a relational database table, the TyphonQL *rename attribute* operation will translate into an *alter table rename column* command in SQL. The data structure will change but the database contents will remain unchanged. In contrast, if the corresponding TyphonML entity is mapped to a MongoDB collection, the TyphonQL *rename attribute* operation will translate into a *rename* operator. The latter will not alter the data structure, but it will rather change documents data in the MongoDB collection, by replacing the old attribute name with the new attribute name. In case of a key-value database, such an attribute renaming operation is more complex to carry out as no specific native operator exists. The compilation of the TyphonQL *rename attribute* command will therefore imply read, update and write operations.

Through the above examples we can observe that, in the general case, the compilation of platform-independent TyphonQL DDL queries may require the use of native, platform-specific DDL and DML operations.

3 General Architecture

In this section we describe how the evolution of a TYPHON polystore schema is carried out within the general TYPHON architecture. The schema evolution and data migration tools (WP6) rely on the general TYPHON approach to evolve the polystore data structures and migrate the polystore database contents. The evolution tools interact with the tools developed in TYPHON work packages such as WP2 (TyphonML editor) and WP4 (TyphonQL compiler). In this section, we also specify the input parameters that the evolution and migration tools need to receive, as well as the functions to invoke from the other TYPHON components.

Our general approach is guided by the entire process of an evolution in a database system, from the specification of what each schema evolution operator exactly means to which artefacts are impacted by this evolution and how to carry out such an adaptation.

Concerning the use case that we considered, it denotes a typical schema evolution which starts with a user that modifies an existing, already deployed TyphonML model. The schema modifications requested by the user are then propagated (when needed and when possible) to the actual polystore (relational or NoSQL) data structures and to the data instances. The user will also be provided with a query transformation tool allowing the migration of existing TyphonQL queries to the target TyphonML schema[1].

3.1 Architecture Overview

As shown in Figure3, a holistic view about the architecture of our tool is shown. The evolution process includes ten main steps. This process has been slightly modified from our previous deliverable (D6.2) in agreement with the other project partners in order to guarantee general consistency and ease the integration of the TYPHON components .

[1]The query migration tool will be developed in the next months, and will be presented in deliverable D6.4.

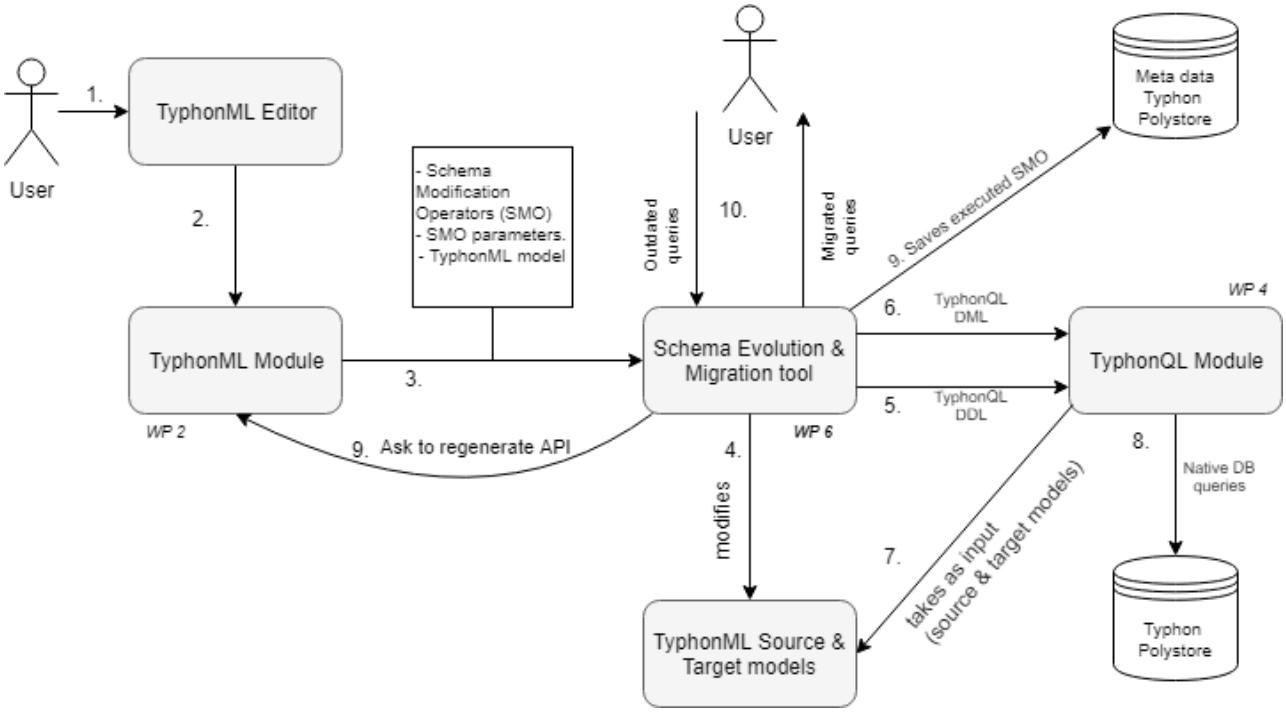

Figure 3: Global architecture and interactions of the evolution and migration tools.

The different steps are detailed as follows:

1. The user edits the TyphonML model using the specific *evolution mode*. He can manipulate *Typhon object* using *Change Operator* defined in the TyphonML meta models (as described in D2.3.). Each of these objects represents one of the Schema Modification Operator (SMO) defined in our previous deliverable (D6.2). (Step 1)

2. Once the step 1 is completed, the model is saved as a .tml file (i.e., extension file). The TyphonML module then produces an .xmi file that can be easily read by programs of the other work packages. (Step 2)

3. The TyphonML module uses the global polystore API in order to call the evolution tool. This invocation is issued by means of the *evolve* function, and takes the source TyphonML and TyphonDL models as input parameters. The TyphonDL model is used to verify that the physical structure needed are indeed already present in the current running TyphonDL. This is a precondition to all evolution operators, if the corresponding structure does not exist the evolution will be rejected. (Step 3)

4. The evolution tool receives the TyphonML model as an .xmi file. It reads it and extracts all the Change Operators requested by the user. Each operator is recorded with the date and TyphonML model it applies to. This allows the developer to easily retrieve all executed changes in order to apply them on a similar TyphonML model running on a different environment.

5. The evolution tool applies the evolution operators and adapt the different polystore artefacts.

    - After verifying the validity of the operator the TyphonML model is adapted to the model resulting from the application of the schema change operator. (Step 4)

- One may also need to modify the platform-dependent database structure, for example adding a new table, or a new collection. This is done by generating and executing the corresponding TyphonQL DDL function(s) starting from the source TyphonML model (Step 5 & 7).
- Finally the related data is migrated to comply with the the target TyphonML model. This also consists of TyphonQL DML queries executed on a given TyphonML model. (Step 6 & 7)

6. The TyphonQL engine developed in Work Package 4 receives the TyphonQL DML and DDL queries and is in charge of compiling them into native database queries for data structure and contents manipulation. (Step 8)

7. Once all TyphonML change operators are executed they are saved to an external database and discarded from the model by the evolution tool.

8. Finally the schema evolution tool calls back the TyphonML module to inform it that the evolution process is complete and that the current polystore schema can be changed to the target TyphonML model. If needed, the TyphonML module will eventually regenerate the Data Access API available to the client applications. The adaptation of the client applications to the new data access API is out of the scope of the TYPHON project.

3.2 User interaction with the TyphonML editor

Let us now recall how the schema modification operators will be expressed by the user through the TyphonML editor.

We have identified two different ways for the user to express schema evolution changes, namely *implicit* and *explicit*.

- *Implicit*: The user freely edits the TyphonML model during an *evolution session*. At the end of the session, the TyphonML tool computes the difference between the source TyphonML model (the state of the model at the start of the evolution session) and the target TyphonML model (the state of the model at the end of the evolution session). From this difference the TyphonML tool derives a list of modification operators that have been performed. This has the advantage of being transparent for the user. However, it can be challenging to record all model editing changes (including *undo's*), and to derive a correct, minimal list of schema modification operators in all possible cases.
- *Explicit*: The TyphonML editor provides the user with an exhaustive list of Schema Modification Operators (SMOs). When she wants to modify the TyphonML model she has to explicitly choose one of the available evolution operators. Some operators require input parameters that the user will also define in the TyphonML editor. Obviously, the user may repeat this operation several times, by applying several successive schema changes to the TyphonML model.

As already mentioned in deliverable D6.2, together with the other TYPHON R&D partners, we have chosen the **explicit** way of expressing selection schema changes. The output result of this explicit selection will be a list of SMOs that have to be propagated to the actual polystore data structures, to the data instances, and (optionally) to existing TyphonQL queries. The TyphonML module will send this list of operators to the schema evolution and migration tool, that will take care of the propagation operations.

Figure 4 shows the graphical representation in the evolution mode of the TyphonML editor for three *Change Operators*.

Figure 5 represents the same Change Operators in the textual representation of the TyphonML model

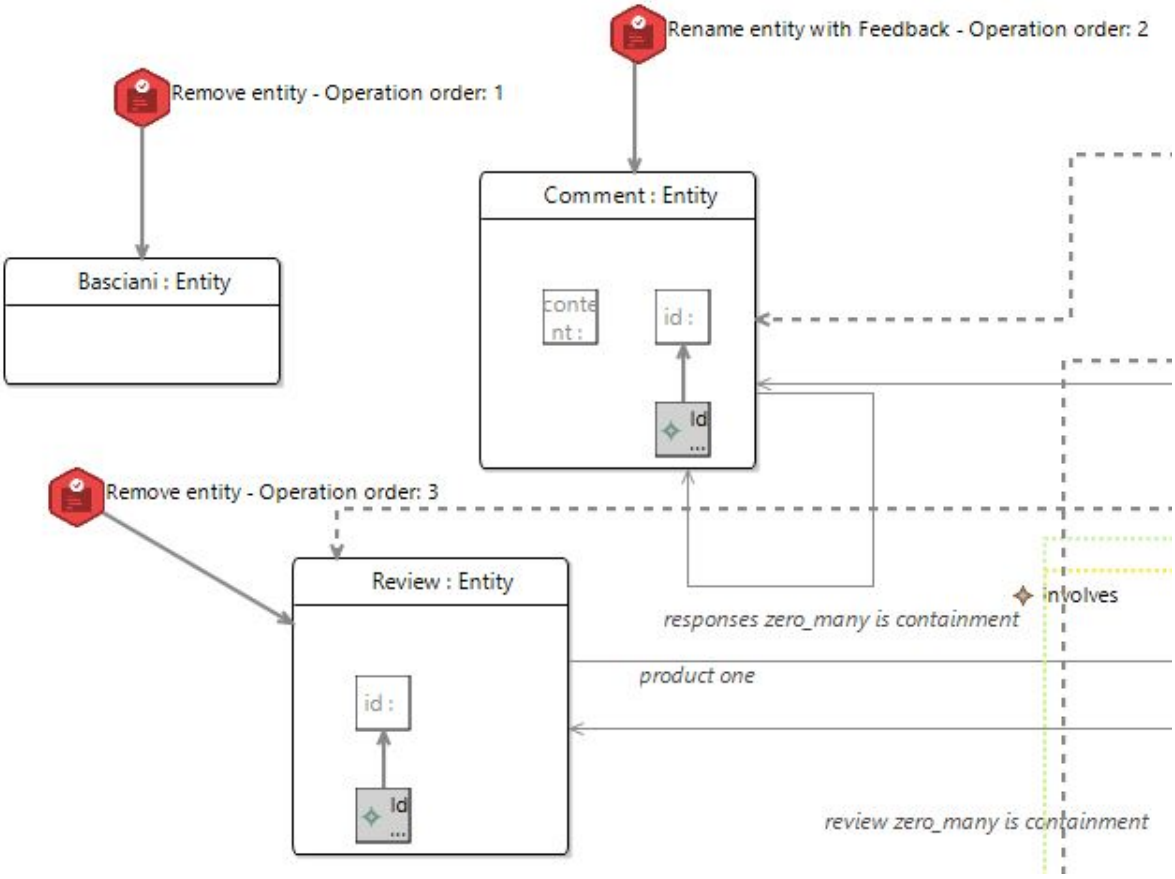

Figure 4: Change Operators in TyphonML Editor.

```
//...
entity CreditCard{
    id : String
    number : String
    expiryDate : Date
    identifer (id)
} entity Basciani { }


relationaldb RelationalDatabase{

documentdb DocumentDatabase{

changeOperators [ remove Entity Basciani
                  rename Entity Comment as Feedback
                  remove Entity Review
                ]
```

Figure 5: Change Operators in TyphonML text representation.

3.3 Schema modification operators

The polystore schema evolution operators that are supported by TYPHON have been described in detail in deliverable D6.2. This list of operators includes, among others, *add entity*, *remove attribute*, *migrate entity*, etc. Each of these operators requires specific input parameters in order to propagate the evolution to the different impacted artefacts.

The evolution and migration tool implements the propagation of the different schema evolution operators available to the user from the TyphonML editor. It provides a generic *evolve* function through a REST API. This function takes as input a list of Schema Modification Operator objects. For each SMO object, the schema evolution and migration tool will verify that the parameters are correct before executing the evolution operators.

3.4 Source and target TyphonML models

A TyphonML model includes two main parts that belong to two different levels of abstraction. The higher level corresponds to the conceptual level, where entity types are described together with their attributes and with the relationships that hold between them. This high-level representation is platform-independent. The lower level belongs to a logical level of abstraction, where the conceptual entities are mapped with their actual database representation (tables, collections, etc.). This level defines on which database system (relational database, document-based, key-value store, etc.) each conceptual entity will be persisted.

For supporting the evolution process at least the current (source) TyphonML model and the target TyphonML model (resulting from the application of the schema evolution operators to the source model) must be available as global variables. The target model will be an SMO parameter sent by the TyphonML module to the evolution and migration tool. The TyphonQL module will also take both source and target TyphonML models as (implicit) inputs to compile the DDL and DML queries sent by the schema evolution and migration tool to propagate the TyphonML schema changes.

For example, when migrating a TyphonML entity *E* from a relational database table to a MongoDB collection database, the evolution tool will send a *read* TyphonQL query on (expressed on the current TyphonML model M_1) to the TyphonQL module. This *read* query will be compiled into a SQL *select* statement. Then, the evolution tool will send a TyphonQL *write* query, expressed on the target TyphonML model M_2, where E is linked to the MongoDB collection. This *write* statement will therefore be compiled into a MongoDB *insert* statement and finally write on the related MongoDB collection. Detailed evolution scenarios that further illustrate this mechanism will be illustrated in Section 6.

3.5 Interaction with TyphonQL

In the general case, the schema evolution and migration tool will need to use two instances of TyphonQL query executor: one instance executing the TyphonQL queries on top of the current (source) TyphonML model, and one instance executing the TyphonQL queries based on the target TyphonML model.

The TyphonQL queries will be generated by the schema evolution and migration tool from the received list of SMOs and their input parameters. For propagating a given SMO to the the polystore data structures and contents, several TyphonQL queries may be generated and executed.

For instance, splitting an entity into two entities requires (1) a TyphonQL DDL query to create an new entity with its corresponding migrated attributes, (2) several TyphonQL DML queries to migrate the data from the source entity attributes to the target entity attributes through *read-write* operations, (3) several TyphonQL DDL queries to delete the migrated attributes from the source entity.

3.6 Data access API regeneration

Finally the schema evolution tool calls back the TyphonML module to inform it that the evolution process is complete and that the current polystore schema the polystore can be changed to the target TyphonML model. If needed, the TyphonML module will eventually regenerate the Data Access API available to the client applications. The adaptation of the client applications to the new data access API is out of the scope of the TYPHON project.

In Section 6 we illustrate the use of our schema evolution and data migration tool, by considering a concrete cross-platform evolution example: the migration of a TyphonML entity from a database platform (relational) to another (MongoDB).

4 Design

In this section we present the design of the evolution and migration tool, as well as the global workflow that is triggered when the TyphonML model including the list of requested schema changes is received via a call to the generic *evolve* function. We also present the interactions with the components developed in the other work packages and the different API functions involved in the process.

4.1 Internal function classes

Let us first describe below the different internal action classes defined in the evolution tool. Figure 6 shows the model and main classes that are involved when treating a TyphonML *Change Operator*.

- RestController. This class is the interface between the evolution tool and other components of the TYPHON polystore. It exposes the *evolve* function that takes two parameters. The first parameter is a path to the source TyphonML model and the second parameter is a path for the saving of the final TyphonML model. This function will read the source TyphonML model, extract the *Change Operators* and convert each of them to the internal representation of a Schema Modification Operator (SMO) of the evolution tool.
- EvolutionToolFacade. This class receives the extracted list of *SMOs* and the object representation of the TyphonML model. Its responsibility is to ask the evolution tool to execute each SMO and then to save to the given target path the final resulting TyphonML model.
- EvolutionToolService. This class corresponds to the core implementation of the evolution tool. It consists of a series of specific functions, each corresponding to a given evolution operator. Those functions are called by the EvolutionToolFacade. They take as parameters an SMO object that contains the required information to execute the evolution operator and the TyphonML representation object on which to apply this operator. A new Model object is finally returned with the evolution changes applied. This class and its methods are also in charge of propagating schema changes to the other polystore artefacts such as the TyphonML model, the data, the platform-dependent structures and, in the near future, to TyphonQL queries.
- TyphonMLInterface. All functions manipulating the TyphonML model are provided by this interface. There are functions reading the model and making it easier to get specific Typhon objects such as *getEntityFromName(String entityName);* and there are writing functions that create or adapt objects in the TyphonML model.
- TyphonQLInterface. This interface provides high level functions to manipulate data and structures. In the end they produce and execute TyphonQL DDL and DML queries.
- TyphonQLConnection. This interface is defined in deliverable D4.2. It provides functions allowing one to execute TyphonQL queries. TyphonQL DML queries return results using the WorkingSet format which is also defined in deliverable D4.2.

4.2 Internal Data Objects

This section describes the internal data objects used by the evolution and migration tool. Each objects contain the attributes that the EvolutionService needs in order to evolve the schema, the data and data structures. We implemented an Adapter pattern between our own data objects and the core objects of the TyphonML library for two main reasons. First, the development of the evolution tool was done in parallel with the development

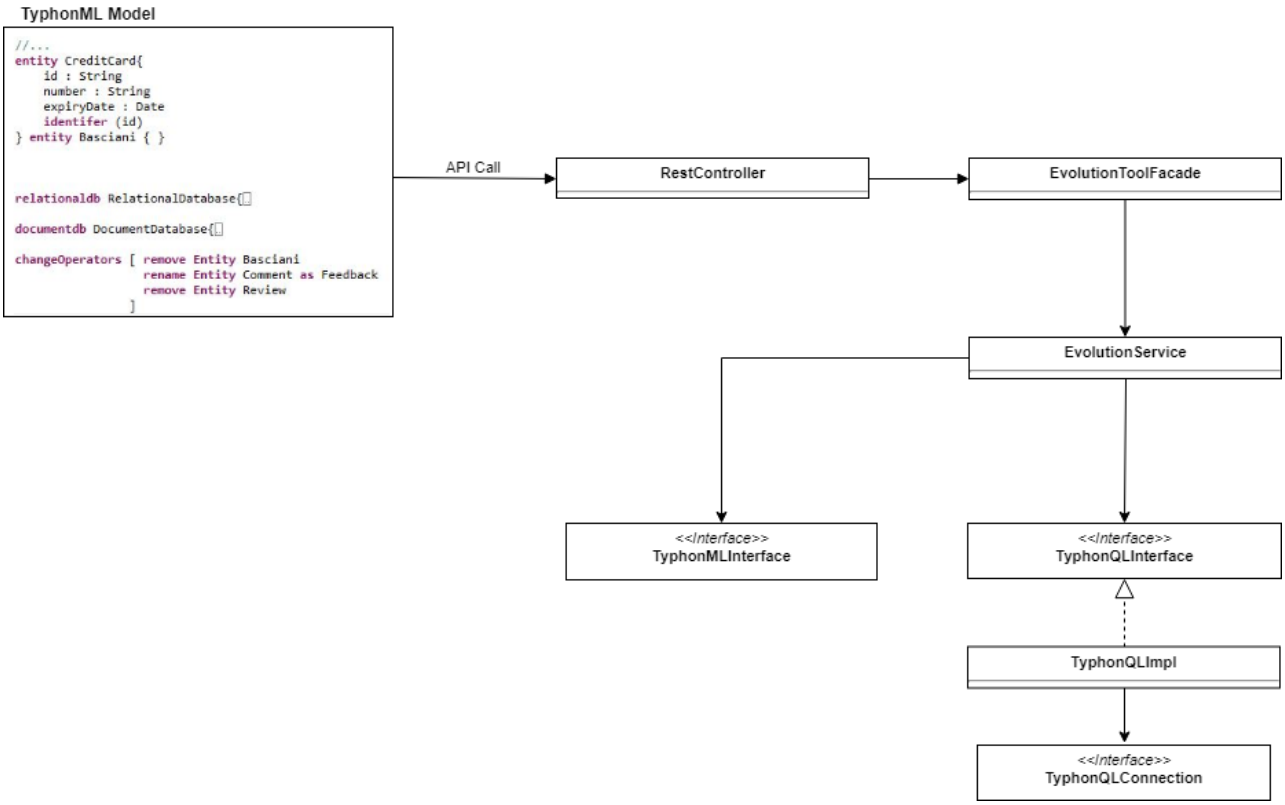

Figure 6: Main classes of the evolution tool.

of the TyphonML library. It was therefore more convenient to use interfaces that fit our requirements. Second, this provides independent implementation of the evolution operator in the EvolutionService class. In case of change in the core objects of the TyphonML library, only the adapters will have to be modified.

- EvolutionService. This class constitutes the core implementation of the evolution operators. It makes the link between the internal data objects and the other modules' interfaces described in section 4.1.
- SMO. This class is the internal object representing an evolution operator, it is the equivalent of $ChangeOperator$ in the TyphonML model. It contains the type of Typhon object, the specific evolution operator that must be applied, as well as the user parameters required to evolve the polystore. SMOs have been specified in detail in our previous deliverable D6.2.
- EntityDO. This class represents an TyphonML Entity Type. It contains the name, a map of the attributes and the identifier.
- RelationDO. The relation domain data object requires information about the source and target Entities, that are represented using EntityDO domain object as well. *Containment, opposite* and *cardiniality* are the same information as defined in the TyphonML model.
- SMOAdapter. This class implements the SMO interface. It takes a $ChangeOperator$ (defined by the TyphonML library) as constructor parameter. Input parameters required by the EvolutionService interface are set in the SMO object based on the TyphonML $ChangeOperator$ attributes.
- EntityAdapter. Similarly to the SMOAdapter class. It implements $EntityDO$ interface and allows one to convert Entity TyphonML library objects.
- RelationAdapter. This class adapts Relation TyphonML library object to our $RelationDO$ data object.

- ChangeOperator. This class represents an evolution operator in the TyphonML meta model. This is the object produced by the user when she uses the TyphonML editor in evolution mode as described and illustrated in Figure 5.
- Model. The TyphonML library object representing the complete TyphonML model. This class is also used as parameter of any evolution operator in the EvolutionService class.
- Entity. The TyphonML library Entity class.
- Relation. The TyphonML library Relation class.

For more detailed information about the TyphonML library, please refer to deliverable D2.3.

4.3 Input Parameters

In Table 1 we list the different parameters that are required by Schema Modification Operators supported by the evolution tool. Those parameters are needed to propagate the changes to the TyphonML model, the data structures and the data instances. Section 5.2 will list and describe those parameters with their corresponding Schema Modification Operators.

Table 1: List of user input parameters for each supported SMO.

Parameter Name	**Parameter type**	**Description**
Attribute	Attribute	An attribute object with name and datatype.
AttributeName	string	Name of an existing attribute.
DatabaseName	string	Name of a database, used to specify the mapping of a newly created entity.
DatabaseType	string	The type of database (relationaldb, documentdb,...), used to create a correct mapping in the TyphonML model.
Entity, FirstEntity, SecondEntity	EntityDO	An EntityDO object used to create a new type in the TyphonML model.
EntityName	string	Name of an entity.
isOuterJoin	boolean	Used in merge SMO, true or false if the data result is an outer join.
JoinAttributeFirst	string	Used in merge SMO, name of the join attribute in the first entity.
JoinAttributeSecond	string	Used in merge SMO, name of the join attribute in the second entity.
MergeEntityName	string	Name of the resulting merged entity.
NewEntityName	string	New name of the renamed entity.
Relation	RelationDO	A RelationDO object used to create a new relation.
RelationName	string	Name of a relation.

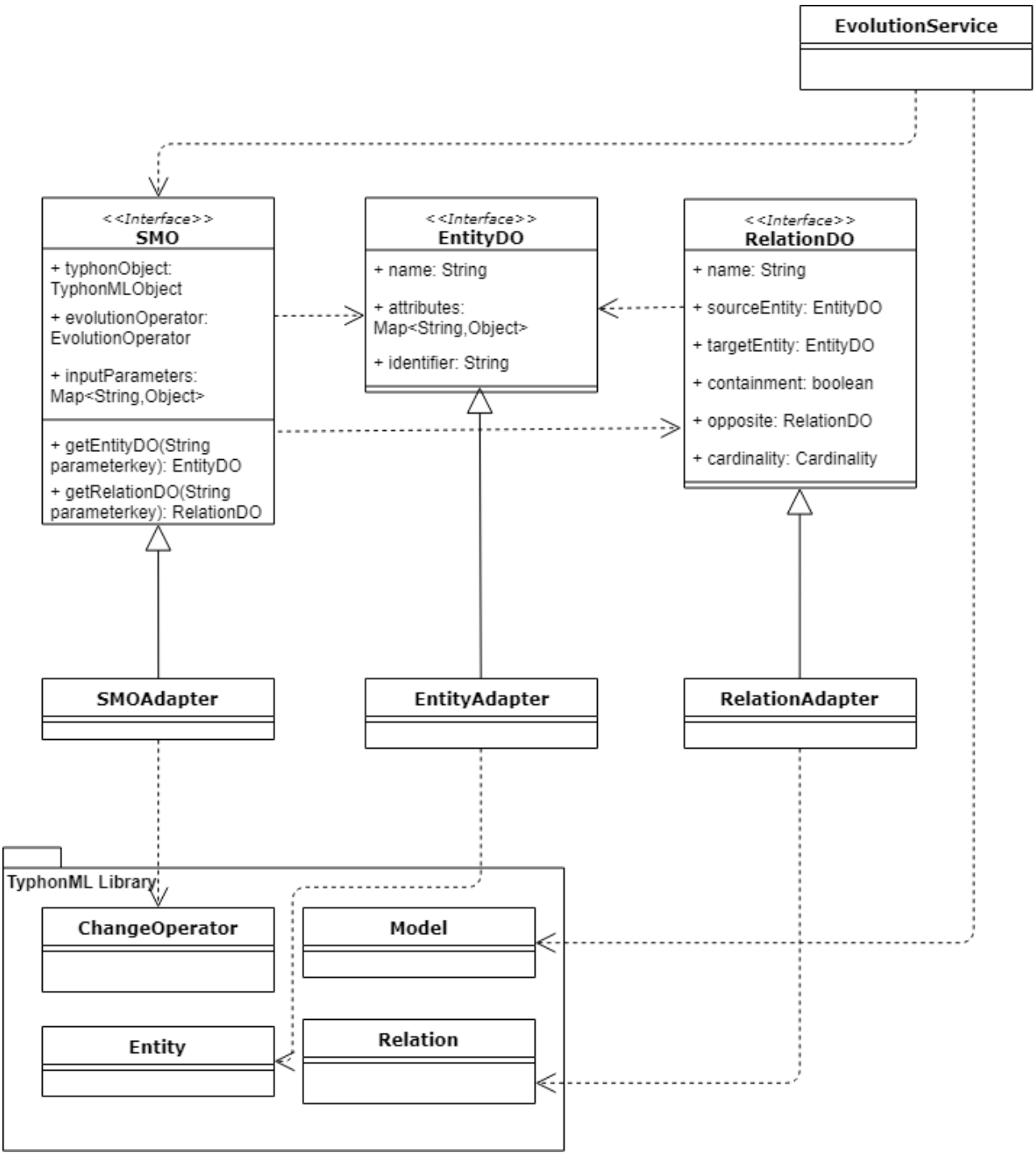

Figure 7: Main classes of the evolution tool domain objects.

4.4 Interfaces to other work packages

We list and describe in this section the interfaces that manipulate other TYPHON artefacts, namely the TyphonML model, the platform-dependent data structures and the database contents via the TyphonQL query

language. The manipulation of the model is entirely handled by the evolution tool itself. In contrast, the adaptation of data structures and database contents required by the evolution operators are implemented by means of the TyphonQL module (WP4). The evolution tool interacts with the TyphonQL engine by sending DDL and DML queries issued by its TyphonQLInterface.

4.4.1 TyphonMLInterface

The TyphonMLInterface specifies functions to manipulate a Model object representing the full TyphonML model. There are two types of methods in this interface: *query methods* that allow to query the model easily, and *manipulation methods* that actually change the model. Query methods return particular TyphonML objects inside the model, e.g., one can get an Entity, a Relation or verify that a certain Entity is not involved in any relation. All these methods take a TyphonML Model object as parameter either to query it, to copy or to modify it. Those methods are :

- Entity getEntityTypeFromName(String entityname, Model model); This method returns a TyphonML library $Entity$ object with name $entityname$.
- **boolean** hasRelationship (String entityname, Model model); Returns true if the Entity of the given name $entityname$ is involved in a relation. This is used to prevent a deletion of entity type still used by others.
- DatabaseType getDatabaseType(String entityname, Model model); Returns the DatabaseType TyphonML library object of the Entity with $entityname$.
- Relation getRelationFromNameInEntity(String relationname, String entityname, Model model); Returns the Relation corresponding to the given $relationname$ in the Entity with given $entityname$.
- Database getDatabaseFromName(String dbname, Model model); Returns a Database object with the given name $dbname$.
- String getDatabaseName(String sourceEntityName, Model model); Returns the name of the Database where the given Entity with name $sourceEntityName$ is mapped.

The methods manipulating the Model are self-describing and are listed in Figure 8.

4.4.2 TyphonQLInterface

The methods in this interface receive high-level objects and are in charge of compiling them into corresponding TyphonQL DML queries for reading and writing data. Those methods can also call the TyphonQL DDL methods that allow one to modify data and structures depending on the operation and the source TyphonML model.

TyphonQL Data Manipulation Language functions The functions listed below are data manipulation functions used by the evolution tool. They all eventually call a TyphonQL DML query as specified in deliverable D4.2 , *query, insert or delete*.

- WorkingSet readAllEntityData (String entityId , Model model); Retrieves all entity data, all attributes of entity named $entityId$. This is compiled using the source $model$ provided. The implementation uses the TyphonQL *query* function.
- WorkingSet readEntityDataEqualAttributeValue (String sourceEntityName, String attributeName , String attributeValue , High level function that reads data of a given entity and applies an equal where condition on given attribute name with the given attribute value. It uses again the *query* function.

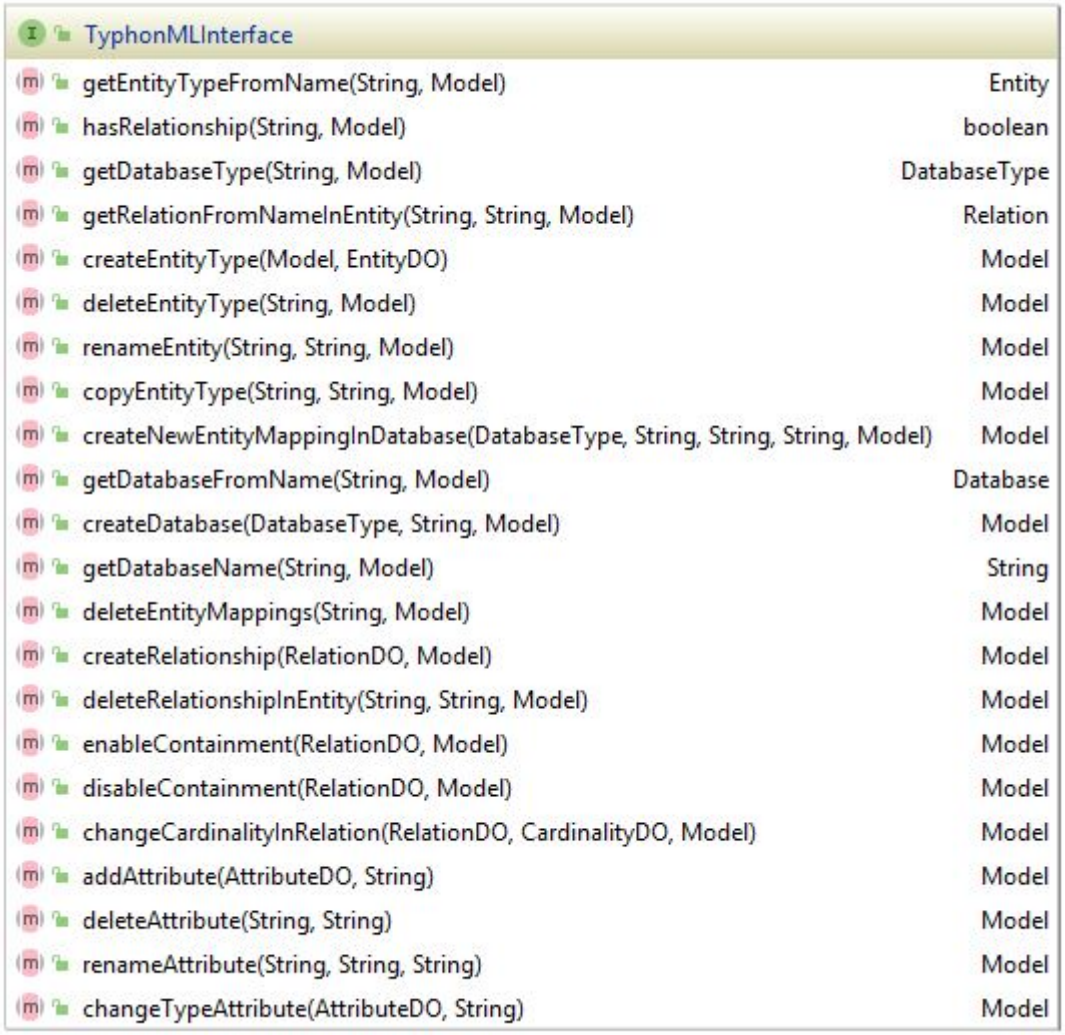

Figure 8: TyphonMLInterface definition.

- WorkingSet readEntityDataSelectAttributes (String sourceEntityName, List<String> attributes , Model model); Reads only the data of selected $attributes$ from the entity $sourceEntityName$.
- WorkingSet deleteAllEntityData (String entityid , Model model); Deletes all data in Entity named $entityid$ on given TyphonML $model$. Uses TyphonQL *delete* function (D4.2).
- **void** deleteWorkingSetData(WorkingSet dataToDelete , Model model); Deletes the data contained in the given Working Set $dataToDelete$.
- **void** writeWorkingSetData(WorkingSet workingSetData, Model model); Writes the given Working Set $workingSetData$. Using *insert* TyphonQL function.
- WorkingSet readRelationship (RelationDO relation , Model model); Reads data related to the given relationship $relation$.

TyphonQL Data Definition Language functions The functions listed below are data definition language (DDL) functions. The TyphonQL compiler is in charge of compiling the invocations of those functions into corre-

sponding data structure changes and/or data adaptations. The resulting native database operations depend on the underlying database the evolving TyphonML objects are mapped to.

- **void** createEntityType (EntityDO newEntity, Model model); Calls the TyphonQL DDL function that creates a new Entity $newEntity$. It will eventually create a new table (Relational model) or create a new collection (document model) for entity $newEntit$ in the polystore.
- **void** renameEntity(String oldEntityName, String newEntityName, Model model); Renames the entity $newEntityName$ to $oldEntityName$. This can lead to renaming of logical table or collection.
- **void** deleteEntityStructure (String entityname, Model model);Asks TyphonQL to delete the structure of an entity type.
- **void** deleteAttributes (String entityname, List<String> attributes , Model model); Deletes data and structures of the given attribute list in the given entity type.
- **void** createRelationshipType (RelationDO relation , Model model); Depending on the underlying databases. Creates foreign key (for relational) or changes the way the data must be inserted (for NoSQL).
- **void** deleteRelationship (RelationDO relation , **boolean** datadelete , Model model); Deletes the relationship. Implies reference attribute deletion or foreign key constraint deletion for relation data model.
- **void** enableContainment(String relationName, String entityname, Model model); For NoSQL databases it implies the replacement of the reference attribute with the full target entity object. This operator is not available for relational databases.
- **void** disableContainment(String relationName, String entityname, Model model); Deletes data in the source entity and replace it with a reference attribute to the target entity data.
- **void** changeCardinalityInRelation (String relationname , String entityname, CardinalityDO cardinality , Model model);

4.5 Schema Modification Operators

Table 2 lists all the Schema Modification Operators (SMOs) that can be performed. We list the TyphonML object and the evolution operators that can be applied to it. We then provide a brief description of the SMO and of its preconditions, if any. $variableNames$ are the input parameters requested for the operator. They are described in the implementation Table 3 and in the input parameter Table 4.3. Next are two columns indicating if the operator has an impact on data and/or on data structures. More details about the operators are available in deliverable D6.2.

Typhon Object	Evolution Operator	Description	Preconditions	Impact on structure	Impact on data
Entity Type	**Add**	A new entity is created in the TyphonML model and the corresponding data structure is created in the polystore.	An instance of the target database is deployed by TyphonDL.	X	-
Entity Type	**Remove**	The entity type is removed from the schema. Any mapped database object is also removed from the polystore (data structure and instances are deleted).	The schema includes an entity with the given name. This entity has no relation with other entity types.	X	X
Entity Type	**Rename**	The entity $oldEntityName$ is renamed to $newEntityName$. If the corresponding polystore object (table, collection) has the same name as $oldEntityName$, it is also renamed as $newEntityName$.	The $sourceModel$ schema includes an entity named $oldEntityName$.	X	X
Entity Type	**Split Horizontal**	Migrates the instances of entity $sourceEntityName$ that has a given value $attributeValue$ of their attribute $attributeName$ to a new entity $targetEntityName$ mapped to a database type $DatabaseType$ on $DatabaseName$ in structure $TargetLogicalName$	$sourceModel$ includes entity $sourceEntityName$. $Databasename$ of type $Databasetype$ is running.	X	X
Entity Type	**Split Vertical**	Migrates the instances of entity $Entity$ to two new entity $FirstNewEntity$ and $SecondNewEntity$. A new one-to-one relationship $RelationName$ is created between the two.	$Databasename$ of type $Databasetype$ is running.	X	X
Entity Type	**Merge**	$FirstEntity$ and $SecondEntity$ are merged into one new entity named $MergeEntityName$. It's done based on an equal join condition on attribute in first entity named $JoinAttributeFirst$ and $JoinAttributeSecond$ in the second entity. $isOuterJoin$ is set to true if the not matched lines of $SecondEntity$ must also be included in the merged entity.	$FirsEntity$ and $SecondEntity$ must be linked via a relation. $SecondEntity$ must not be in more than one relation.	X	X

Entity type	**Migrate**	Migrates the data structure and instances of entity $entityName$ to another database platform $databasetype$ with name $Databasename$ on a logical structure named $Targetlogicalname$	$Databasename$ of type $Databasetype$ is running.	X	X
Relationship Type	**Add**	A relationship type between two entity types is added.	Both TyphonML entities linked by $relation$ have explicitly declared identifiers. The current data instances of those entities respect the cardinality constraints of $relation$. If isContainment is set to true, and $relation$ is not a one-to-one relationship, then the referencee entity is not mapped to a relational database (no denormalized data in relational databases).	X	
Relationship Type	**Remove**	A relationship between two entity types is removed. Data structures and instances are adapted accordingly		X	X
Relationship Type	**Rename**	The relationship $relation$ is renamed as $newName$.	No existing relation with $newName$ exists in the same source Entity.	X	X
Relationship type	**Enable containment**	Enabling the TyphonML attribute of a relation, which specifies that the target entity is contained in the source entity. This means that the data instances of the referenced entity are stored in the referencee data structure.	Source entity must have an attribute containing the reference identifier to the target entity. If $relation$ is not a one-to-one relationship, then the referencee entity is not mapped to a relational database (no denormalized data in relational databases).	X	X

Relationship type	**Disable Containment**	Disabling the containment. The target entity becomes standalone and references between the entities are added, following the direction of the relationship and depending on the underlying database platform. In a relational database, the target entity references the source entity. In a document-oriented database, the source entity references the target entity.	The target entity has a declared identifier.	X	X
Relationship type	**Enable opposite**	Creates a new relationship $relationName$ that is the opposite of the given $relation$, source becomes target, and target becomes source, cardinality is also reversed.	Existing data is not in conflict with the new relation.	X	X
Relationship type	**Disable opposite**	This deletes the opposite relation of the given $relation$.	$Relation$ exists.	X	X
Relationship type	**Change Cardinality**	The cardinality between two entities is modified. The maximum and minimum cardinalities can increase or decrease.	The data already conforms to the new cardinality constraints.		
Attribute	**Add**	An attribute $Attribute$ is added to a TyphonML Entity with $Entityname$ as name.	The entity has no attribute with the same name as $Attribute$.	X	-
Attribute	**Delete**	An attribute $Attribute$ is deleted from an entity with $Entityname$, i.e., its structure and data instances.	The deleted attribute is not member of an index nor of an identifier.	X	X
Attribute	**Rename**	Attribute is renamed.	$AttributeName$ exists in $Entityname$ entity.	X	X
Attribute	**Change type**	The data type of the input attribute is changed.	The changed type cannot be an Entity type. This is supported by the Add Relationship SMO.	X	X

Table 2: Specification of the Schema Modification Operators.

5 Implementation

In this section we present how the Schema Modification Operators are implemented in the EvolutionService class. We first describe the generic high-level operations that we perform for each evolution operator. We then present a summary table exposing the specific implementation details of each operator .

5.1 Generic operations

Once the EvolutionService class SMO operation function is called by the EvolutionFacade class, it goes through a series of operations that are generic to each SMO :

1. $VerifyInputParameters$. We verify that the given SMO contains the necessary input parameters in order to perform the corresponding evolution operations. For instance the creation of a new entity requires a valid Entity object as well as information about the database mapping.
2. $TyphonMLmodification$. We proceed to create a copy of the given TyphonML model. Using the TyphonMLInterface implementation class (described in 4.1) we modify this copy according to the requested evolution of the model. The resulting adapted model is then returned to the calling function.
3. $TyphonQLDDLStructurechanges$. We call the TyphonQL interface corresponding to structure editing functions. This will issue a TyphonQL DDL query that will create or edit the physical structure (creation of a table or collection, creation of attributes, modification of a type,...).
4. $TyphonQLDMLReaddata$. We read the data using a TyphonQL DML query on a TyphonQL engine running on a given TyphonML model, which in this case is the source model given as input.
5. $TyphonQLDMLWritedata$. We may edit the $WorkingSet$ data retrieved in order to fit the target data model. This data is then written in the database using writing functions of the $TyphonQLInterface$. A TyphonQL engine running on the target TyphonML produced at step three is initiated, and we feed it with a writing query containing the (adapted) working set.
6. $TyphonQLDMLDeletedata$. We delete the migrated data instances that fit the source TyphonML model but are not valid anymore.
7. $TyphonQLDDLDeletestructure$. If needed we delete the obsolete structures (tables, columns,...) using TyphonQL DDL functions.
8. $Returnfinalmodel$. The final TyphonML model is returned to the EvolutionFacade where it can be used as input model of subsequent evolution operators.

5.2 Specific SMO operations

In this section we present Table 3 which details, for each Schema Modification Operator, the different input parameters required as well as the different operations that will be executed.

The first two columns $TyphonObject$ & $Evolutionoperator$ specify the TyphonML object and the evolution operation, respectively. Together they uniquely identify the evolution operation to execute. Those attributes are derived from the different Change Operator classes defined in TyphonML library. The *SMOAdapter* is in charge of the variables initialization based on the$ChangeOperator$ object.

The third column *Input Parameter* lists the input parameters needed to execute the SMO. Those parameters are user-specified when applying a Change Operator in the TyphonML editor (once in evolution mode). Table 1 provides more details about those inputs.

TyphonML operations column lists all the operations applied on the TyphonML model. Those functions are part of the *TyphonMLInterface* (see 4.4. They are used to query the model for verification and mainly to add, modify or delete objects. It then produces the final model resulting of the application of the evolution operator.

TyphonQL DDL column lists functions that use the TyphonQLInterface to produce TyphonQL DDL statements. Those may manipulate the data structures and the data instances depending on the underlying database model. The TyphonQL compiler is in charge of translating these operations to specific native database statements. See Section 4.4.2 for a detailed description of those functions.

TyphonQL DML lists the operations that requires data manipulation by the evolution tool. It reads particular data using a TyphonQL DML query and optionally modifies the Working Set (the object representing the data returned by TyphonQL engine) and it finally writes queries using a modified TyphonML model that will be used in order to produce correct native queries.

Typhon Object	Evolution Operator	Input Parameter	TyphonML operations	TyphonQL DDL	TyphonQL DML
Entity Type	**Add**	Entity DatabaseName Databasetype	createEntityType createEntityMappping	createEntityType	-
Entity Type	**Remove**	Entityname	hasRelationship deleteEntityType	deleteEntityStructure	deleteAllEntityData
Entity Type	**Rename**	OldEntityName NewEntityName	renameEntity	renameEntity	-
Entity Type	**Split Horizontal**	SourceEntityName TargetEntityName TargetLogicalName DatabaseType DatabaseName AttributeName AttributeValue	copyEntityType createEntityMappping	-	ws = readEntityDataWith-EqualAttribute writeWorkingSetData (modified ws) deleteWorkingSetData(ws)
Entity Type	**Split Vertical**	Entity FirstEntity SecondEntity DatabaseName Databasetype RelationName	createEntityType(FirstEntity) createEntityType(SecondEntity) createRelationship deleteEntityType	createEntityType createRelationshipType	readEntitySelectAttributes writeWorkingSetData
Entity Type	**Merge**	FirstEntity SecondEntity MergeEntityName JoinAttributeFirst JoinAttributeSecond isOuterJoin	getCardinality deleteRelationshipInEntity hasRelationship addAttribute renameEntity	addAttribute deleteRelationship renameEntity	readRelationship writeWorkingSetData
Entity type	**Migrate**	EntityName DatabaseName Databasetype Targetlogicalname	deleteEntityMapping createEntityMappping	createEntityType deleteEntityStructure	readAllEntityData writeWorkingSetData deleteWorkingSetData
Relationship Type	**Add**	Relation	createRelationship	createRelationship	
Relationship Type	**Remove**	Relationname Entityname	deleteRelationshipInEntity	deleteRelationshipInEntity	
Relationship type	**Enable containment**	Relation	getDatabaseType enableContainment	enableContainment	
Relationship type	**Disable Containment**	Relation	disableContainment	disableContainment	

Relationship type	**Enable opposite**	Relation RelationName	createRelationship enableOpposite	createRelationship	
Relationship type	**Disable opposite**	Relation	deleteRelationshipInEntity	deleteRelationshipInEntity	
Relationship type	**Change Cardinality**	Relation Cardinality	changeCardinalityInRelation	changeCardinalityInRelation	
Attribute	**Add**	Attribute Entityname	addAttribute	addAttribute	
Attribute	**Delete**	Attribute Entityname	removeAttribute	removeAttribute	
Attribute	**Rename**	Attributename Entityname NewNameAttribute	renameAttribute	renameAttribute	
Attribute	**Change type**	Attribute Entityname	changeAttribute	changeAttribute	

Table 3: Parameters needed and operations executed by SMO.

6 Migrate Entity Scenario

In this section we will illustrate in further details the execution of the Migrate Entity operator. This operator will be instantiated in the context of a concrete example. We will show the changes applied to the TyphonML model, the propagating operations performed on the polystore data structures and instances using TyphonQL.

Before modification: initial state of the polystore Let us start with the current TyphonML model. In our example, this current model consists of several entity types. One entity type, $User$, is mapped to a relational database $MySQL$ table called $clientsTable$. Another one $Order$ is mapped to a document database called $MongoDB$. This is shown in the TyphonMl Editor in Figure 9.

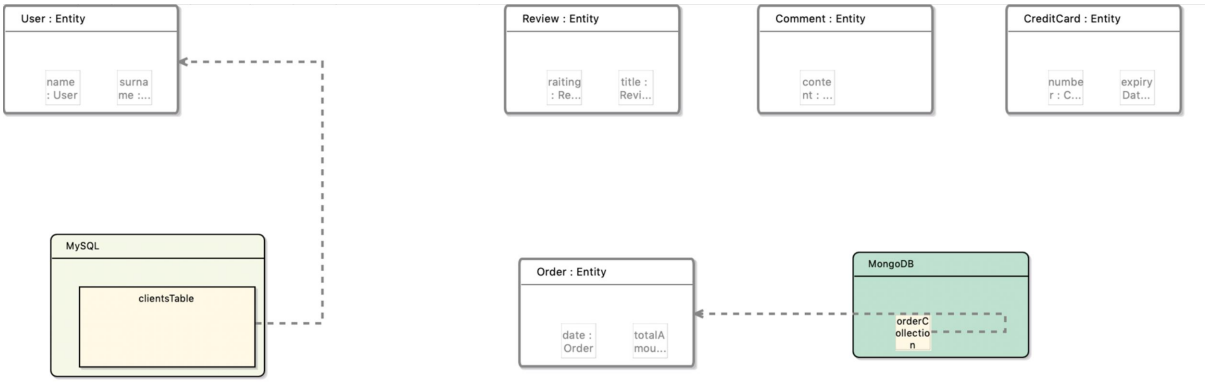

Figure 9: Initial TyphonML model.

During modification: intermediate state of the polystore Let us now assume that the user decides - via the TyphonML editor evolution mode- to apply the $migrate$ operator on entity $User$, in order to migrate the $User$ instances towards a collection in a MongoDB database $MongoDB$. Figure 4 shows examples of Change Operators in this evolution mode.

The TyphonML module then calls the schema evolution and migration tool via the *evolve* function of $RestController$ (see 4.1) and passes the TyphonML model including the $migrate$ Change Operator (as in figure 5.

The schema evolution and migration tool first reads the TyphonML model .xmi file and loads the Change Operators. Then $EvolutionToolFacade$ uses $SMOAdapter$ class to transform $ChangeOperator$ objects into SMO objects. Each one of them is then sent and the corresponding function is called in $EvolutionService$. It verifies the parameters given as input of the function. We also check that the current running TyphonDL indeed contains the requested databases. If those preconditions are not met the evolution is declined and notified to the user. If everything is correct the operator is applied, as detailed in Table 3 with functions in $TyphonMLInterface$ and $TyphonQLInterface$ being executed. We illustrate this execution with a log trace shown in Figure 10.

The first phase is the adaptation of the TyphonML model. Line 3-4 are in charge of creating the new mapping of $User$, from $clientsTable$ to $userCollection$.

The second phase consists of modifying the structures, the existing $MongoDB$ document database must contain a $userCollection$ collection. This is the role of the TyphonQL DDL generated at lines 5-6.

The third phase concerns the adaptation of the data instances. The schema evolution and migration tool reads the data from the entity type $User$ using the TyphonQL query running on the initial TyphonML model $initial_Model$ (line 7) into $UserDataWorkingSet$. This data is then rewritten to $userCollection$ collection of the document database $mMongoDB$ using the TyphonQL write statement on the produced TyphonML model $final_Model$ (line 8).

The last steps are the deletion of data in $clientsTable$ using the TyphonQL DML $delete$ function (line 9), and the deletion of the table itself (lines 10-11).

```
1  [main] INFO com.typhon.evolutiontool.services.EvolutionServiceImpl - Verifying
       input parameter for [ENTITY] - [MIGRATE] operator
2  [main] INFO ...TyphonDLInterfaceImpl - Verifying that database [MongoDB] of type
       [documentdb] is running
3  [main] INFO ...TyphonMLInterfaceImpl.deleteEntityMapping - Delete database
       mapping of entity type [User]  in TyphonML
4  [main] INFO ...TyphonMLInterfaceImpl.createDatabase - Creating a mapping
       Database [MongoDB] of type [DOCUMENTDB] to entity [User] mapped to [
       userCollection] in TyphonML
5  [main] INFO ...TyphonInterfaceQLImpl.createEntity - Create entity [User] via
       TyphonQL DDL query on TyphonML model [initial_Model]
6  [main] INFO ...TyphonQLConnectionImpl - Executing TyphonQL DDL [TQLDDL CREATE
       ENTITY User (name, surname)] on TyphonML [final_Model]
7  [main] INFO ...TyphonQLConnectionImpl - from User e select e on TyphonML [
       initial_Model]
8  [main] INFO ...TyphonQLConnectionImpl - TyphonQL 'insert' command working set :
       UserDataWorkingSet on TyphonML [final_Model]
9  [main] INFO ...TyphonQLConnectionImpl - TyphonQL 'delete' command working set :
       UserDataWorkingSet on TyphonML [initial_Model]
10 [main] INFO ...TyphonInterfaceQLImpl.deleteEntity - Delete entity [User] via
       TyphonQL DDL on TyphonML model on TyphonML [initial_Model]
11 [main] INFO ...TyphonQLConnectionImpl - Executing TyphonQL DDL [TQLDDL DELETE
       ENTITY User on TyphonML [initial_Model]
```

Figure 10: Log of the Migrate Entity operator scenario.

After modification: final state of the polystore The final state of the polystore is shown in Figure 11. The $Client$ entity instances are still in the polystore, but they are now stored in a MongoDB collection. All existing TyphonQL queries accessing the $Client$ entity instances remain valid. They will now be compiled into MongoDB statements (e.g., $find$) instead of SQL queries (e.g., $select$).

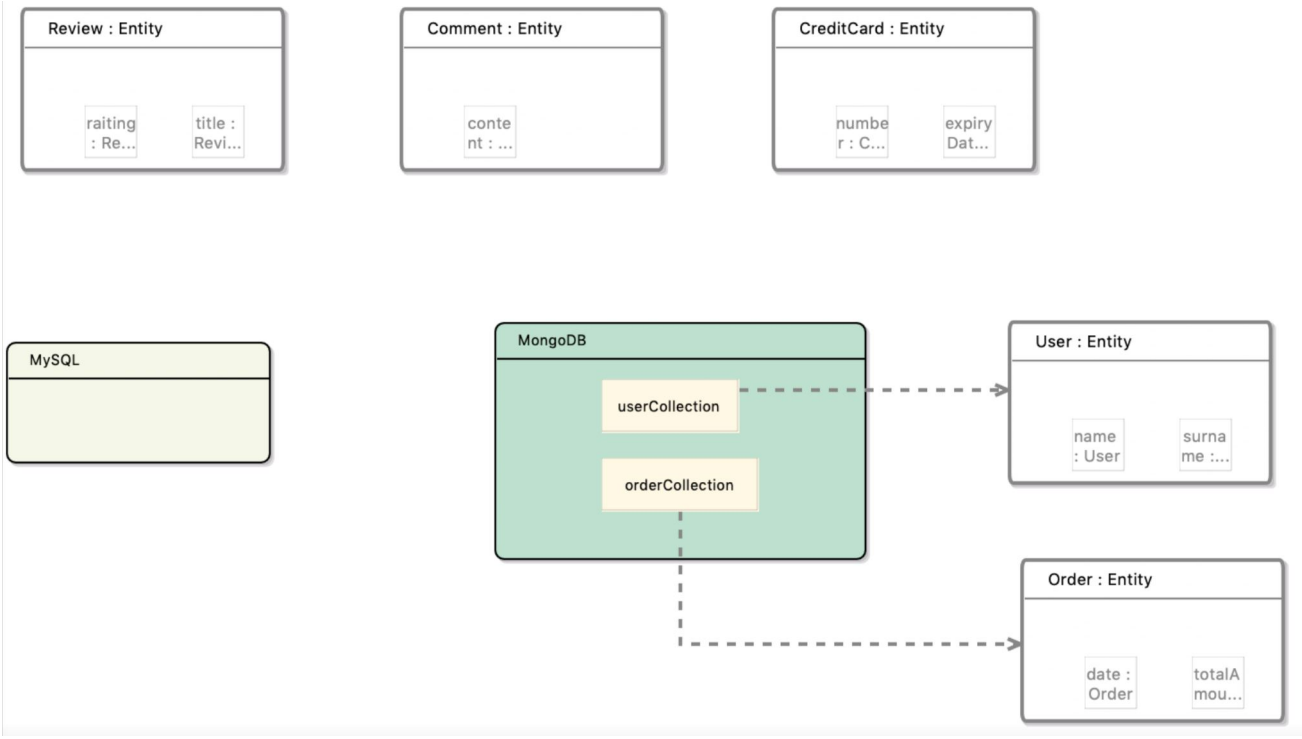

Figure 11: Final state of the polystore, after applying the Migrate Entity operator.

7 Conclusions

In this deliverable, we presented the tools developed to support schema evolution and related data migration in TYPHON hybrid polystores. We particularly focused on the general architecture of those evolution and migration tools, their design and their implementation. We described the way the schema evolution operators specified in deliverable D.6.2, are now implemented in the TYPHON WP6 component using a generic evolve function that can be triggered from the evolution mode of the TyphonML editor. The schema evolution and data migration tools currently support (1) the adaptation of the TyphonML model from a list of schema evolution operators, (2) the adaptation of the underlying platform-specific data structures by exploiting the TyphonQL DDL operators, (3) the migration of the data from the source to the target schema,

The next steps in WP6 include, among others: (1) the full integration of the schema evolution and migration tools with the other TYPHON components, namely the TyphonML editor (WP2) and the TyphonQL engines (WP4); (2) the development of the query migration tool; (3) the development of recommendation techniques and tools that would automatically suggest polystore reconfigurations to the user based on the continuous monitoring of the polystore; (4) systematic testing and evaluation of the WP6 components in collaboration with the use case partners, (5) the dissemination of our research and development results to a wide audience.