Project Number 780251

D2.3 Hybrid Polystore Modelling Language (Final Version)

Version 1.0
22 December 2018
Final

Public Distribution

University of L'Aquila

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 Konstantinos Papathanasiou Themistocleous 87 106 83 Athens Greece Tel: +30 210 344 7300 E-mail: kpapathanasiou@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	1 August 2018
0.2	First draft	23 September 2018
0.7	First full draft	1 December 2018
0.8	Further editing draft	15 December 2018
1.0	QA review	22 December 2018

Table of Contents

1 Introduction **1**

1.1 Structure of the deliverable 1

2 Validation of the TyphonML language **2**

2.1 Validation with respect to the Use case requirements 2

2.2 Validation with respect to technical requirements of WP2 7

2.3 Summary of the requirements not satisfied by the initial version of TyphonML 9

3 The revised TyphonML language **10**

3.1 Relational database concepts 12

3.2 Document database concepts 13

3.3 Key-value database concepts 13

3.4 Graph database concepts 14

3.5 Column database concepts 15

3.6 TyphonML change operators 16

3.6.1 Change operators for conceptual elements 16

3.6.2 Change operators for logical elements 18

3.6.3 Use of the TyphonML change operators 20

3.7 Enabling natural language processing 20

4 Hybrid Polystore Access Infrastructure and API **22**

4.1 A microservice-based architecture for the Polystore API 22

4.2 The Data Access Layer Generation Process 24

4.3 Managing data relationships 28

4.3.1 One-to-Many relationships 29

4.3.2 Many-to-Many relationships 29

5 Conclusions **32**

Executive Summary

Relational database management systems (RDBMS) have become over the years the predominant choice for storing large volumes of data. As such, various techniques and tools have been developed to support their design and development. In recent years, NoSQL databases have emerged as an alternative approach to data storage, lauded for their horizontal scalability and flexibility. NoSQL database systems have come a long way, however they still remain far from the level of maturity of relational databases. While there is some work towards this direction, the proposed solutions are technology-specific and not applicable across different classes of NoSQL data stores.

This document presents the final version of the TyphonML language, a new language for modeling in a homogeneous manner (and by abstracting the specificities of the underlying technologies) the data to be stored in polystores consisting of both relational and NoSQL databases.

1 Introduction

The need for levels of availability beyond those supported by relational databases and the challenges involved in scaling such databases horizontally led to the emergence of a new generation of purpose-specific databases grouped under the term NoSQL. In general, NoSQL databases are designed with horizontal scalability as a primary concern and deliver increased availability and fault-tolerance at a cost of temporary inconsistency and reduced durability of data.

Designing and deploying a hybrid data persistence architecture that involves a combination of relational and NoSQL databases, and which can manage different types of structured and textual data, is a complex, technically challenging and error-prone task. Even though several techniques and tools have been introduced to support the design and development of relational systems, standardised notations and supporting tools for designing NoSQL databases are not available yet.

The aim of TYPHON is to provide an industry-validated methodology and integrated technical offering for designing, developing, querying, evolving, analysing and monitoring architectures for scalable persistence of hybrid data (relational, graph-based, document-based, textual etc.).

In the context of TYPHON, WP2 will develop a technical infrastructure for *designing* hybrid polystores taking into account the structure of the data, the availability, partitioning and consistency requirements of different subsets of the data and the available deployment resources. In particular, the main objectives of WP2 are the development of the TyphonML language and supporting tools to model in a homogeneous manner, and by abstracting the specificities of the underlying technologies, the data to be stored.

In this document, we present results of WP2 related to the following task (from the TYPHON DoW):

> *Task 2.1: Hybrid Polystore Modelling Language (TyphonML) Design.* This task will design the abstract syntax of TyphonML, a modular and extensible language for modelling hybrid polystores. TyphonML will provide constructs for hybrid data modelling as well as facilities for modelling availability, consistency and partitioning requirements, and the available infrastructure on which the hybrid polystore will be deployed. The language will be defined through an iterative process consisting of two main steps: a) elicitation of new concepts from the considered application domains, and b) validation of the elicited and properly formalized concepts by modeling concrete hybrid polystores.

In particular, the refinements that have been operated on the initial version of the TyphonML (initially presented in D2.1 [3]) are motivated and presented. The needed refinements have been identified by taking into account both Use Case and WP2 requirements elicited and presented in D1.1 [6].

1.1 Structure of the deliverable

The deliverable is structured as follows:

- Section 2 describes the process that has been followed to refine the initial version of TyphonML.
- Section 3 introduces the final version of TyphonML in terms of its abstract syntax defined by means of EMF-based technologies.
- Section 4 presents an initial implementation of the microservice-based architecture of the hybrid polystore access infrastructure, and corresponding API and generator.
- Section 5 concludes the document and provides an overview of the next steps.

2 Validation of the TyphonML language

In this section we discuss the validation activities we have done on the initial version of the TyphonML language, which was initially structured as shown in Figure 1.

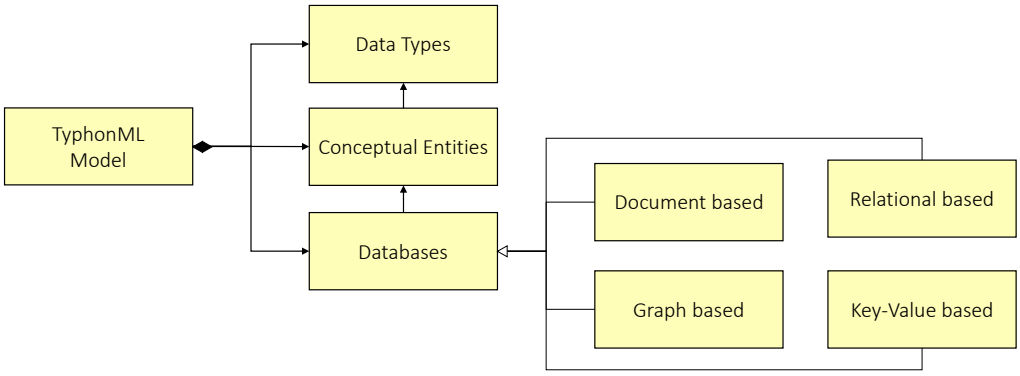

Figure 1: High-level view of the initial version of TyphonML (as presented in D2.1 [3])

Figure 2 shows the TyphonML specification of the data entities to be managed by an explanatory e-commerce system. Such entities are given in a technology independent way with the aim of defining the attributes and the relationships of the conceptual entities that the information system being developed has to manage. Such conceptual entities are then referred by database specific definitions (given in the same TyphonML model) as shown on the right hand side of Figure 2. In the shown TyphonML specification, the data entities *Order*, *User*, *CreditCard*, and *Product* will be stored in a relational database. The *Product* entity will be also managed by a graph based database in order to manage the concordance of customers in buying similar products.

By referring to Figure 3 (initially presented in D2.1 [3]), this document is mainly about the activities *Assessment of the TyphonML expressiveness* and *TyphonML definition/refinement* applied on the first version of the language presented in D2.1. More specifically, by means of the *Assessment of the TyphonML expressiveness* phase, actual polystore specifications are analysed in order to check if there are unforeseen requirements that are not satisfied, or if the language does not provide modelling constructs, which are relevant to represent the system at hand. To this end, the validation has been performed by considering the following sources of information:

- Use Case requirements, defined by the industrial partners of the project (see Table 1)
- WP2 requirements (see Table 2)
- Development of the data access infrastructure and API (see Section 4)

According to the outcome of such a phase, in the *TyphonML definition/refinement* activity new concepts are added, and the identified issues are fixed as discussed in the next section.

2.1 Validation with respect to the Use case requirements

As discussed in D1.1 [6] the industrial Use Case requirements specification is intended to provide a quantitative view of the envisioned TYPHON solution, stating measurable criteria that should be met during the implementation of the research and development tasks within the whole project. In this respect, since TyphonML plays

```
entity Order {
    date : date
    totalAmount : real
    products -> Product."Product.orders" [*]
    paidWith -> CreditCard
    user -> User."User.orders"
}

entity User {
    name : string
    comments :-> Comment [*]
    paymentDetails :-> CreditCard [*]
    orders -> Order [*]
}

entity CreditCard {
    number: string
    expiryDate:date
}

entity Product {
    name: string
    description: string
    photos -> Photo[*]
    reviews :-> Review."Review.product" [*]
    orders -> Order[*]
}

entity Comment {
    content: natural_language
    responses :-> Comment[*]
    annos -> Annotation[*]
}

entity Annotation {
    _from: int
    _to: int
    type: string
    content : string
}

entity Photo {
    _id : string
    image : jpeg
}

entity Review {
    product -> Product
}
```

```
relationaldb myDb{
    table {
        Order:Order
        index OrderIndex { "Order.date" }
        id {"Order.date"}
    }

    table {
        User:User
        index UserIndex { "User.name"}
        id {"User.name"}
    }

    table {
        CreditCard:CreditCard
        id {"CreditCard.number"}
    }

    table {
        Product:Product
        id {"Product.name"}
    }
}

graphdb myGraph {
    node ProductNode:Product{
        name = "Product.name"
    }

    edge {
        from ProductNode
        to ProductNode
        label {
            weight: int
        }
    }
}
```

Figure 2: Specification fragment in TyphonML of the e-commerce system presented in D2.1

a key role in the project, and underpin the data modeled and managed by all the tools developed in the technical work packages, it is necessary to validate the expressiveness of TyphonML with respect to the Use Case requirements. In particular, in D1.1 [6] the following requirement categories are identified:

C1. Polystore modeling language
C2. Polystore modeling tools
C3. Text data modeling
C4. Polystore deployment language
C5. Polystore deployment
C6. Polystore query language
C7. Queries on structured data
C8. Queries on textual data
C9. Data analytics and monitoring

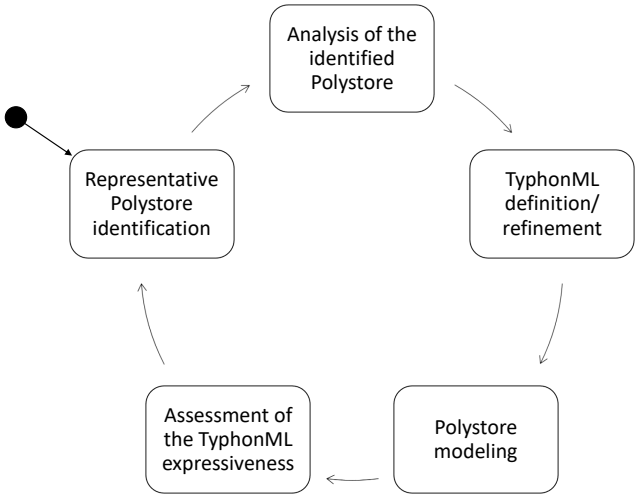

Figure 3: Process to define the TyphonML language

C10. Polystore data migration
C11. Interfaces
C12. General

The requirement categories that are strictly related with the work in WP2 and in particular with the definition of the TyphonML language are C1 and C2. The other requirements categories underpin the work of other WPs.

In the following the TyphonML metamodel defined in D2.1 [3] is discussed with respect to the requirements in C1. The requirements in C2 will be the subject for the work on the supporting modeling tools that we are developing in the context of WP2 and that will be presented in the deliverable *D2.4 – TyphonML Modelling Tools* (due at M18). All the requirements are identified in terms of the modalities SHALL, SHOULD, and MAY defined as follows [6]:

- SHALL is used to denote an essential requirement. A typical system could not be used, would not work, or cannot be validated if this requirement is not fulfilled. SHALL requirements are of highest priority for validation of the platform.
- SHOULD is used to denote a requirement that would help a typical system be easier to use, or to work better, even if it is not essential; in that case a trade-off can be achieved between development costs on the technology side and user benefit on the system side.
- MAY is used to denote a requirement that can lead to a benefit in order to fulfil an additional evaluation criterion or increase the usefulness of the technology. The fulfilment of the requirement is interesting but only in view of available resources and research and development partner interests.

The requirements in the category C1 are shown in Table 1 and in the following they are discussed individually in order to validate the expressive power of the TyphonML language. In particular, the aim of the discussion is to check if the elicited requirements can be satisfied by means of the initial version of the TyphonML language (in this case the requirement is marked with the symbol ✓). The symbol ⊙ is used to mark requirements that cannot be directly satisfied by the initial version of the language and demanded the refinements presented in the next section in order to address them. The symbol ✗ represents requirements that are not supported yet since they are not strictly related to the TyphonML language and that are instead mainly related to the supporting tools that will be finalized later in the project and presented in the next deliverables of WP2.

ID	Requirement	Overall Priority	Earth Observation Priority	Hybrid Bank Priority	Motorway Monitoring Priority	Smart Vehicles Priority
1	The polystore modelling language shall support schema and schema-less database definitions	SHALL	SHALL	SHALL	SHALL	SHALL
2	The polystore modelling language shall support storing sensor data in a relational database	SHALL	N/A	SHALL	N/A	SHOULD
3	The polystore modelling language should support modelling of existing data stores	SHOULD	SHOULD	SHOULD	SHOULD	SHOULD
4	The polystore modelling language shall support storing recorded trend displays	SHALL	N/A	SHALL	N/A	SHOULD
5	The polystore modelling language should be able to support the storing of query results as cases	SHOULD	MAY	SHOULD	N/A	SHOULD
6	The polystore modelling language shall support field types and operations to handle spatial data and perform basic operations for ingestion, querying and filtering	SHALL	SHALL	SHALL	N/A	SHOULD
7	The polystore modelling language shall support a field type that allows to store a Latitude and Longitude values (e.g. "LatLon")	SHALL	SHALL	SHALL	N/A	N/A
8	The polystore modelling language shall support a field type that allows to store a non-geodetic, general X and Y coordinates (e.g. SpatialType)	SHALL	SHALL	N/A	N/A	SHOULD
9	The polystore modelling language shall support a field type that allows to store a bounding box (e.g. "BoundingBox")	SHALL	SHALL	N/A	N/A	N/A
10	The polystore modelling language should provide data types to store binary fields	SHOULD	SHOULD	SHOULD	N/A	SHOULD
11	The polystore modelling language should allow to handle records with binary fields of up to 2 GB	SHOULD	SHOULD	N/A	N/A	MAY
12	The polystore modelling language should support additional spatial types as defined by the Lucene interface	SHOULD	SHOULD	N/A	N/A	N/A
13	The polystore modelling language should implement the spatial data types and related operations as defined by "OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option"	SHOULD	SHOULD	N/A	N/A	SHOULD

Table 1: Use case requirements (as in D1.1 [6])

1 - The polystore modelling language shall support schema and schema-less database definitions (✓) Such a requirement has played a key role during the definition of the TyphonML language specification. As shown in the explanatory example shown in Fig. 2, different kinds of database definitions are supported by the language.

2 - The polystore modelling language shall support storing sensor data in a relational database (✓) The TyphonML language does not impose any constraints about the data to be stored in the system being developed. Thus, modelers can define custom data types (e.g., to specify data coming from different source, including

sensors) and specify in which database systems corresponding data has to be stored (i.e., in relational or NoSQL technologies).

3 - The polystore modelling language should support modelling of existing data stores (✓) The language has been developed in collaboration with the TYPHON use case providers. Thus, the expressive power of TyphonML has been continuously checked in order to enable the specification of their existing data sources.

4 - The polystore modelling language shall support storing recorded trend displays (⊙) To satisfy such a requirement, TyphonML language has to support the definition of custom data types, enabling also the implementation of the corresponding behaviour. This was not completely possible in the previous version of the language as discussed in Section 2.3.

5 - The polystore modelling language should be able to support the storing of query results as cases (✓) Such a requirement can be satisfied by properly defining at the conceptual level, the needed entities and relations that developers want to possibly use to store query results.

6 - The polystore modelling language shall support field types and operations to handle spatial data and perform basic operations for ingestion, querying and filtering (⊙) This requirement is related to the possibility of defining custom data-types. Even though the initial version of the language enabled the syntactically specification of new data types, the definition of the corresponding implementation was not possible yet as described in Section 2.3.

7 - The polystore modelling language shall support a field type that allows to store a Latitude and Longitude values (e.g. "LatLon") (⊙) This requirement is related to the previous one and it was not fully satisfied by the initial version of the TyphonML language.

8 - The polystore modelling language shall support a field type that allows to store a non-geodetic, general X and Y coordinates (e.g. SpatialType) (⊙) The initial version of the language enabled the definition of custom data types like SpatialType. However, the proper management of such data of this type was not possible yet.

9 - The polystore modelling language shall support a field type that allows to store a bounding box (e.g. "BoundingBox") (⊙) This is similar to the previous requirement and was affected by the same limitations of the initial version of the TyphonML language.

10 - The polystore modelling language should provide data types to store binary fields (⊙) The initial version of TyphonML did not provide modeler with native data types enabling the specification and management of binary fields.

11 - The polystore modelling language should allow to handle records with binary fields of up to 2 GB (✗) This requirement is partially related to the previous one. However, limitations about space constraints are not related to the language and refers to the particular technologies that will be used to manage the data that are specified at the TyponML level.

12 - The polystore modelling language should support additional spatial types as defined by the Lucene interface (⊙) The initial version of the language was not expressive enough to specify custom data types enabling some searching optimizations.

13 - The polystore modelling language should implement the spatial data types and related operations as defined by "OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option" (⊙) TyphonML does not provide modelers with any native domain specific data types. Even though the initial version of the language enabled the definition of custom data types, their management was not possible yet.

2.2 Validation with respect to technical requirements of WP2

As previously done, in this section we use the symbol (✓) to identify WP2 requirements (shown in Table 2) that are already satisfied by the initial version of the TyphonML language as presented in D2.1 [3]. The symbol (⊙) is used to identify WP2 requirements that are not satisfied by the initial version of the language, but will be fixed by the improved version of TyphonML as presented in the next section. Additionally, we use the symbol (✗) for requirements that are not related to the TyphonML language but on supporting modeling tools that are planned to be delivered in the forthcoming months of the project.

D1 - TyphonML shall enable the specification of data entities and relationships that will be stored in different and heterogeneous databases (✓) As shown in the explanatory example shown in Fig. 2, the initial version of the TyphonML language already enabled the definition of conceptual entities and the specification of the logical and physical elements devoted to manage the specified entities.

D2 - TyphonML shall enable the specification of data models by means of both textual and graphical syntaxes (✗) This requirement is not related to the TyphonML definition tasks and refers to the supporting modeling

Req. No.	Requirement	Priority	Orig. WP	Resp. WP
D1	TyphonML shall enable the specification of data entities and relationships that will be stored in different and heterogeneous databases.	SHALL	2	2
D2	TyphonML shall enable the specification of data models by means of both textual and graphical syntaxes.	SHALL	2	2
D3	Facilities for generating CRUD APIs from data models specified in TyphonML shall be provided.	SHALL	2	2
D4	Definition of custom data types to be used in TyphonML data models shall be supported.	SHALL	2	2
D5	Specification of data types that are needed for applying text-specific analysis (e.g. text, video, recordings) shall be supported.	SHALL	2	2
D6	The definition of structured data types (e.g. sentences, facts, entities, events) that can be extracted from text and represented in TyphonML shall be supported.	SHALL	2	2
D7	The specification of non-functional requirements that will instruct the deployment and querying of the modelled data models shall be supported.	SHALL	2	2
D8	TyphonML supporting tools shall detect inconsistent data models (e.g. data entities in relational databases that refer to inexistent collections in document-based data models).	SHALL	2	2
D9	TyphonML supporting tools may provide modellers with early feedback about the specified data models (i.e. deployment feasibility of the modelled data with respect to the actual resource availabilities).	MAY	2	2
D10	TyphonML editors should be instructed to resolve inconsistencies in the data schema that might be due to system and data evolutions.	SHOULD	2	6
D11	The data migration tools shall define the list of schema changes that can be automatically managed for coupled evolution goals. Such a catalogue of schema changes will be enforced during TyphonML editing sessions that are devoted only to schema evolution purposes.	SHALL	2	6

Table 2: WP2 requirements (as in D1.1 [6])

tools that modelers can use to develop polystores. However, a textual editor for TyphonML has been already implemented and the graphical one is planned later in the project.

D3 - Facilities for generating CRUD APIs from data models specified in TyphonML shall be provided (⊙) This was not addressed in D2.1 and it has been one of the refinement objectives addressed in this document as described in the next sections.

D4 - Definition of custom data types to be used in TyphonML data models shall be supported (⊙) This was not fully addressed by the initial version of the language. Modelers had the possibility to specify custom data types only syntactically without any means to define the corresponding implementations.

D5 - Specification of data types that are needed for applying text-specific analysis (e.g. text, video, recordings) shall be supported (⊙) Even though primary data types were supported by the language, text-specific ones enabling natural language analysis were not available.

D6 - The definition of structured data types (e.g. sentences, facts, entities, events) that can be extracted from text and represented in TyphonML shall be supported (⊙) This requirement was not addressed at the time of D2.1 [3] delivering. However, it is now addressed as presented in deliverable D2.2 [5], which is delivered at M12 together with this document to present the results of the work lead by Edge Hill University on natural language analysis.

D7 - The specification of non-functional requirements that will instruct the deployment and querying of the modelled data models shall be supported (✗) TyphonML permits modelers to specify in different ways how to store and manage the modeled conceptual entities. However, at this stage we have not investigated yet which non-functional requirements are needed to affect the deployment of the modeled data entities. This aspect will be investigated in collaboration with WP3 and WP4 in the context of *Task 2.4: Analysis and Reasoning on TyphonML Models*.

D8 - TyphonML supporting tools shall detect inconsistent data models (e.g. data entities in relational databases that refer to inexistent collections in document-based data models) (✗) This requirement is related to the previous one and it will be one of the subjects to be investigated in the context of Task 2.4 whose results will be presented in the deliverable *D2.5 – TyphonML Model Analysis and Reasoning Tools*.

D9 - TyphonML supporting tools may provide modellers with early feedback about the specified data models (i.e. deployment feasibility of the modelled data with respect to the actual resource availabilities) (✗) The supporting modeling tools of the TyphonML are planned to be released at M18 as the main objectives of deliverable *D2.4 – TyphonML Modelling Tools*.

D10 - TyphonML editors should be instructed to resolve inconsistencies in the data schema that might be due to system and data evolutions (✗) The fulfilment of such requirement involves a tight collaboration between WP2 and WP6. In particular, the schema and data evolution operators being conceived in WP6 have to be shown to modelers that will trigger them directly from the TyphonML modeling tools. Such an integration will be fully investigated in the context of *D2.4 – TyphonML Modelling Tools* even though initial results have been already obtained as presented in the next sections.

D11 - The data migration tools shall define the list of schema changes that can be automatically managed for coupled evolution goals. Such a catalogue of schema changes will be enforced during TyphonML editing sessions that are devoted only to schema evolution purposes (⊙) The initial version of the TyphonML language was not supporting the specification of model changes with respect to a set of available change operators. The initial version of TyphonML has been refined in order to include them as described in the next section.

2.3 Summary of the requirements not satisfied by the initial version of TyphonML

By considering the discussion given above, the requirements that were not addressed by the initial version of the TyphonML language as presented in the deliverable D2.1 [3] are shown in Table 3, which shows for each requirement in which document its fulfilment is or will be achieved. Most of the requirements are addressed in Sec. 3 of this document.

Req. ID	Req. description	Addressing deliverable
4	The polystore modelling language shall support storing recorded trend displays	D2.3 - Sect. 3
6	The polystore modelling language shall support field types and operations to handle spatial data and perform basic operations for ingestion, querying and filtering	D2.3 - Sect. 3
7	The polystore modelling language shall support a field type that allows to store a Latitude and Longitude values (e.g. "LatLon")	D2.3 - Sect. 3
8	The polystore modelling language shall support a field type that allows to store a non-geodetic, general X and Y coordinates (e.g. SpatialType)	D2.3 - Sect. 3
9	The polystore modelling language shall support a field type that allows to store a bounding box (e.g. "BoundingBox")	D2.3 - Sect. 3
10	The polystore modelling language should provide data types to store binary field	D2.3 - Sect. 3
11	The polystore modelling language should allow to handle records with binary fields of up to 2 GB	D2.3 - Sect. 3
12	The polystore modelling language should support additional spatial types as defined by the Lucene interface	D2.3 - Sect. 3
13	The polystore modelling language should implement the spatial data types and related operations as defined by "OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option"	D2.3 - Sect. 3
D2	TyphonML shall enable the specification of data models by means of both textual and graphical syntaxes	D2.4
D3	Facilities for generating CRUD APIs from data models specified in TyphonML shall be provided	D2.3 - Sect. 4
D4	Definition of custom data types to be used in TyphonML data models shall be supported	D2.3 - Sect. 3
D5	Specification of data types that are needed for applying text-specific analysis (e.g. text, video, recordings) shall be supporte	D2.3 - Sect. 3
D6	The definition of structured data types (e.g. sentences, facts, entities, events) that can be extracted from text and represented in TyphonML shall be supported	D2.2
D7	The specification of non-functional requirements that will instruct the deployment and querying of the modelled data models shall be supported	D2.5
D8	TyphonML supporting tools shall detect inconsistent data models(e.g. data entities in relational databases that refer to inexistent collections in document-based data models	D2.4, D2.5
D9	TyphonML supporting tools may provide modellers with early feedback about the specified data models (i.e. deployment feasibility of the modelled data with respect to the actual resource availabilities)	D2.4, D2.5
D10	TyphonML editors should be instructed to resolve inconsistencies in the data schema that might be due to system and data evolutions	D2.4
D11	The data migration tools shall define the list of schema changes that can be automatically managed for coupled evolution goals. Such a catalogue of schema changes will be enforced during TyphonML editing sessions that are devoted only to schema evolution purposes	D2.3 - Sect. 3

Table 3: Summary of the requirements that were not addressed by the initial version of TyphonML as presented in D2.1 [3]

3 The revised TyphonML language

In this section the refined version of the TyphonML language is presented. A bird eye view of the language is shown in Fig. 4 and the main additions/refinements are depicted as boxes with darker color. In particular, in order to address the requirements that were not met by the initial version of the language, the language has been refined by adding modeling constructs enabling the specification of *i)* changes to be operated on existing TyphonML models, *ii)* custom datatypes, and *iii)* types enabling the application of natural language analysis techniques. In order to make this document self-contained, in this section we report also the constructs of the TyphonML metamodel that have not been subject to any change since the initial version. The new metamodel elements that have been added during the second stage of the language design process are presented in Section 3.6 and Section 3.7. Refinements have been also operated to the *CustomDataType* metaclass as discussed in the following.

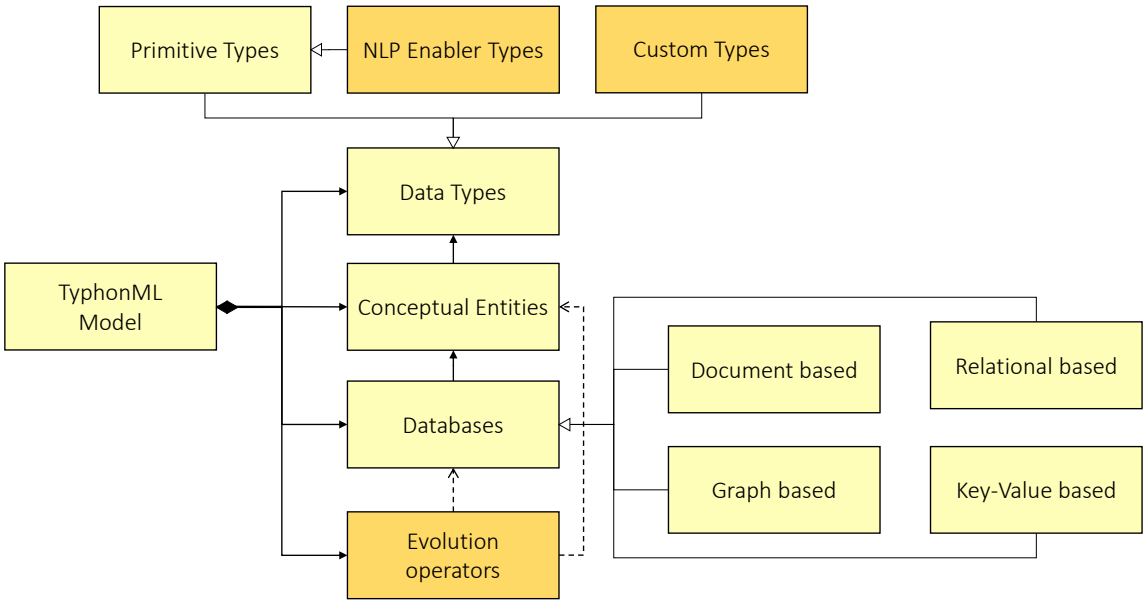

Figure 4: High-level view of the refined TyphonML

The metaclasses implementing the high-level view in Figure 4 are shown in Figure 5 and are described below.

Model This represents the root container of each TyphonML specification, which in turn consists of two element sets. In particular, a model element consists of the following structural features:

- *dataTypes*: it permits to specify all the data types that are used by the system being modeled. As described in the following, the language permits to specify both primitive and complex data types including the conceptual entities to be stored;
- *databases*: it permits to represent all the databases that will be used to actually store the conceptual entities in a polystore infrastructure;

PrimitiveDataType It permits to represent primitive data types like string, data, integer, real, etc.

CustomDataType It extends the abstract *DataType* metaclass in order to enable the specification of custom data types. To this end, each *CustomDataType* instance consists of different elements, which overall contribute the definition of the new data type being defined. For instance, in order to represent geographical points of

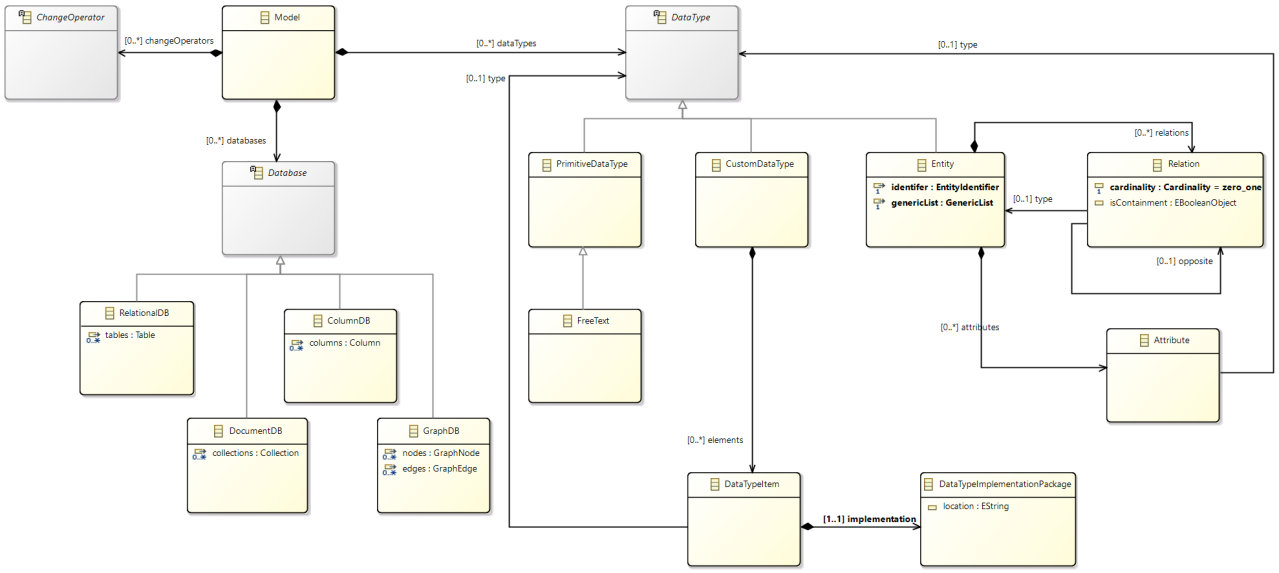

Figure 5: Main metaclasess of the TyphonML metamodel

interest, users can define a *GPS_location* type consisting of two *DataTypeItem elements*, i.e., latitude and longitude. Such data items would be of primitive types (e.g., real). With respect to the initial version of the language, this metaclass has been refined in order to give the possibility to specify the software components/-packages that actually implement the data being defined. For instance, the custom data type defining GPS locations should refer also to the Java classes implementing the getter, setter methods, which are specific to the data type being defined. To this end, a *CustomDataType* can refer to different *DataTypeItem*s.

DataTypeItem It is used to refer the software component/package implementing a new custom datatypes. To this end, each *DataTypeItem* refers to a *DataTypeItemPackage* instance, which contains the location of the software package to be considered.

Entity It plays a key role in the language, since it permits to specify the conceptual entities be managed by the information system being developed. Each entity is defined by means of *attributes* and *relations* (see the metaclasses *Attribute* and *Relation*, respectively).

Attribute It is a named element, which is defined in terms of the type of elements to be represented. As shown in Figure 5, the *type* structural feature is typed *DataType* and consequently it can be a *primitive*, *custom*, or even *entity* type.

Relation It is a named element, which permits to specify relationships between different entities. In particular, the structural features of such modeling constructs are the following:

- *type*: it permits to define the type of the relationship being specified;
- *cardinality*: entities can be involved in relationships of different cardinalities, which can be singular or multiple;
- *opposite*: when creating a reference from one entity (e.g., named *e1*) to a second entity (e.g., named *e2*) it is possible to specify the opposite reference from *e2* to *e1* in order to define a bidirectional relation instead of two different unidirectional ones.
- *isContainment*: it is a boolean attribute, which permits to specify if the target entity is contained (e.g., to trigger cascade-deletion) or not in the entity being modeled.

Database It is an abstract concept, which is specialized in order to specifically represent different kinds of database systems. The currently supported specializations are *RelationalDB*, *DocumentDB*, *KeyValueDB*, *ColumnDB*, and *GraphDB* as detailed in the following subsections.

ChangeOperator It is an abstract concept, which is specialized with concrete evolution operators as detailed in Section 3.6.

3.1 Relational database concepts

The TyphonML concepts enabling the specification of relational databases are shown in Figure 6 and singularly described in the following.

RelationalDB It permits to enhance the conceptual elements defined by means of the modeling constructs previously presented with the aim of specifying which entity should be stored in a relational database and how. To this end, modelers will define the *tables* that are needed to store the entity of interests (see the metaclass *Table* below).

Table It is a named element, which is defined in terms of the following structural features:

- *entity*: it is a reference that permits modelers to specify the data entity that need to be stored in the relational database being specified;
- *indexSpec*: in order to improve the performance of the queries to be evaluated on the relational database being developed, it is possible to adopt indexing mechanisms. In this respect, the language permits to specify the attributes of the table being specified that should be indexed (see the metaclass *IndexSpec*);

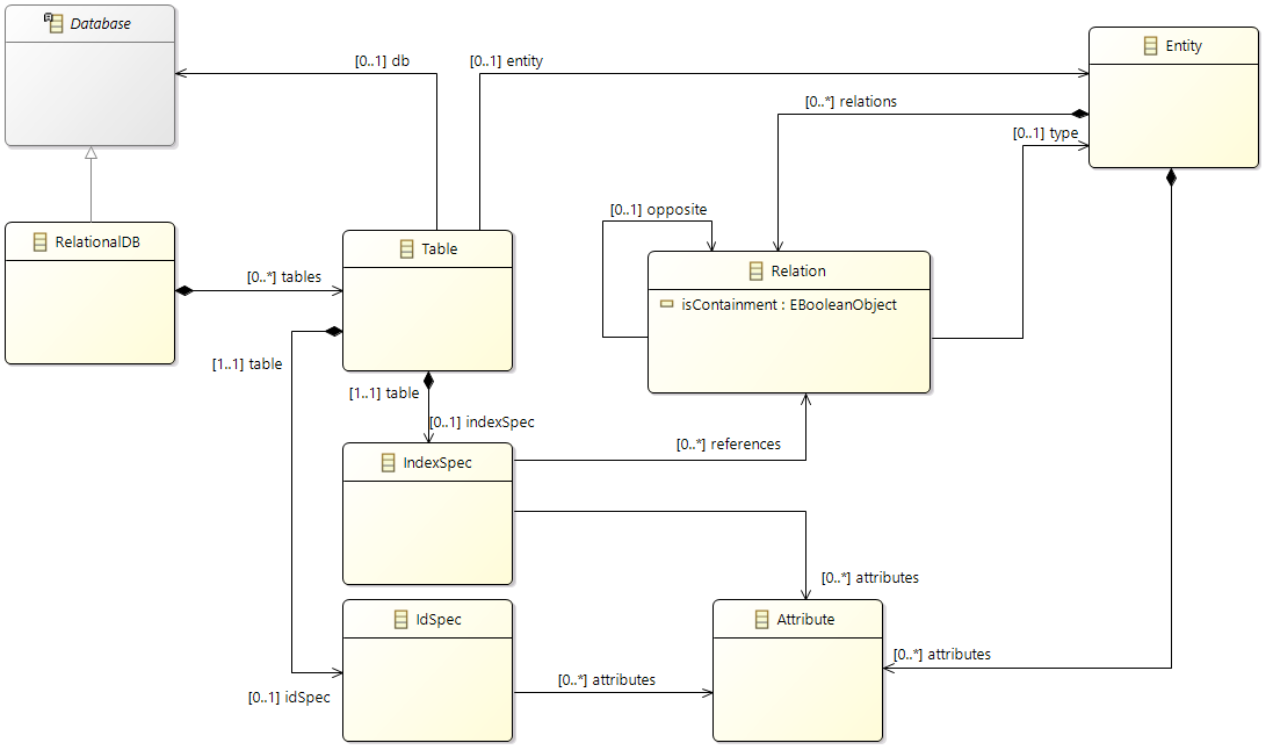

Figure 6: Relational database concepts

- *idSpec*: such a reference is used to define the list of attributes that permit to identify elements in a unique way (see the metaclass *IdSpec*).

IndexSpec Such a metaclass permits modelers to specify the attributes and the references that are part of the index to be created for improving the performance of query executions.

IdSpec It is the metaclass that enables the specification of the table identifiers, which consist of references to entity attributes.

3.2 Document database concepts

The constructs of the TyphonML metamodel that enable the specification of document databases are shown in 7 and described below.

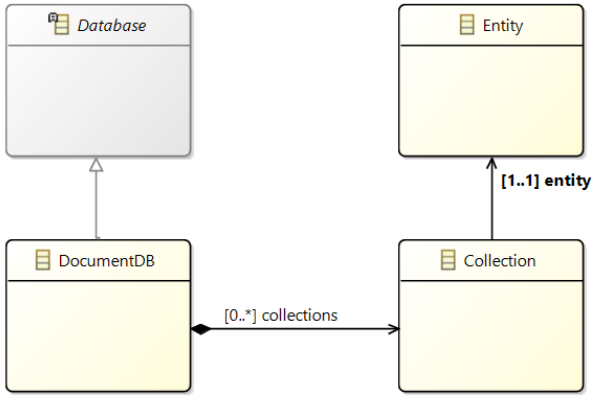

Figure 7: Document database concepts

DocumentDB As summarized in the previous section, document databases are used to store *collections* of heterogeneous elements. Such a metaclass consists of the reference *collections* in order to specify the data entities that each collection should store.

Collection Each collection is devoted to store the data of the referred conceptual data *entity*;

3.3 Key-value database concepts

As discussed in the previous section, key-value stores consist of sets of key-value pairs with unique keys. To this end the metaclasses shown in Figure 8 have been defined.

KeyValueDB It permits modelers to specify aggregations of elements to be stored in key-value pairs.

KeyValueElement Such a metaclass consists of the following two structural features:

- *key*: it is a string attribute to store the key of the pair being specified;
- *values*: it is *DataType* typed reference of multiple cardinality. Thus, the content of the pair being modeled can be an aggregation of heterogeneous data even related to different conceptual entities (see the specializations of the *DataType* metaclass shown in Figure 5).

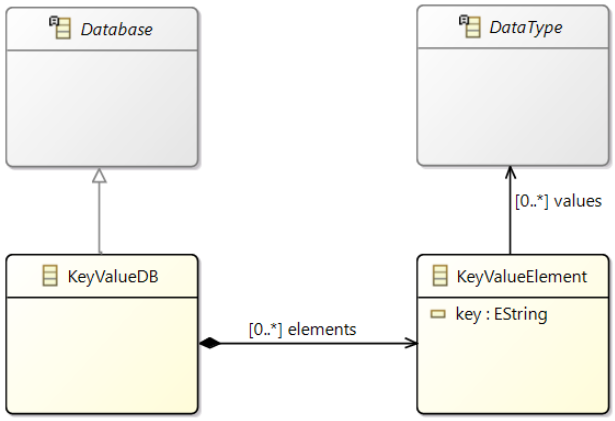

Figure 8: Key-value database concepts

3.4 Graph database concepts

In order to enable the specification of graph databases, TyphonML provides modelers with the modeling constructs shown in Figure 9 and described below.

GraphDB It is a specialization of the abstract *Database* metaclass and permits modelers to specify how the data of interest should be stored in a graph-like structure. Thus, the structural features of the metaclass are the following:

- *nodes*: it is a containment reference used to create nodes in the data structure being modeled;

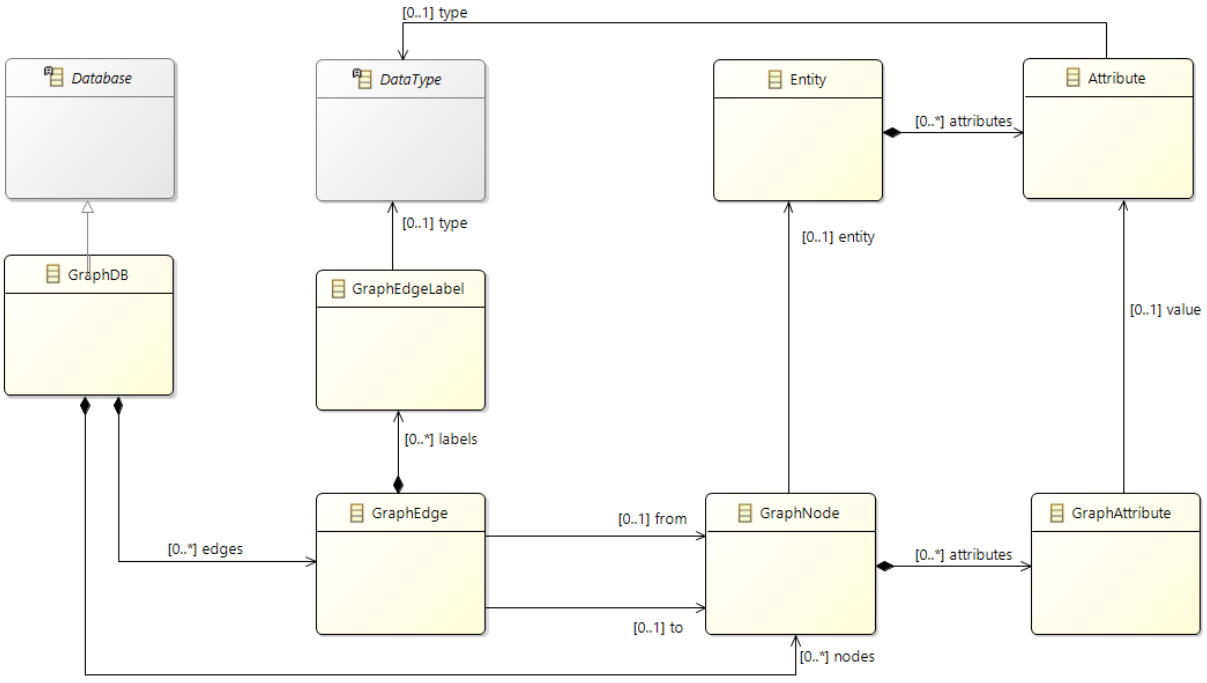

Figure 9: Graph database concepts

- *edges*: it is a containment used to create edges connecting nodes in the data structure being modeled;

GraphNode It permits modelers to specify the nodes of the data structure being modeled and consists of the following structural features:

- *entity*: it is a reference to be used to specify the conceptual entity that has to be managed by means of the graph database being specified;
- *attributes*: further than conceptual entities, modelers might want to specify additional attributes to be stored in each node of the graph.

GraphAttribute As previously mentioned, modelers can specify node attributes and retrieve their values from attributes of conceptual entities (see the reference *value* in Figure 9 between the *GraphAttribute* and *Attribute* metaclasses).

GraphEdge It consists of the structural features that are needed to specify the source and target nodes of the edges being specified. Additional labels can be also defined. In particular,

- *from*: it is the reference that permits to specify the starting node of the edge being modeled;
- *to*: it is the reference that permits to specify the target node of the edge being modeled:
- *labels*: it is a containment reference to create *GraphEdgeLabel* elements defined below;

GraphEdgeLabel It permits to define labels to be attached to edges of the graph-based structure being modeled. Each label is a named element and consists of the corresponding type that can be primitive, custom, or even an entity type (see in Figure 5 the specializations of the metaclass *DataType*).

3.5 Column database concepts

As summarized in the previous section, columns databases permit to store data by column. To this end, TyphonML provides modelers with the *ColumnDB* and *Column* metaclasses shown in Figure 10.

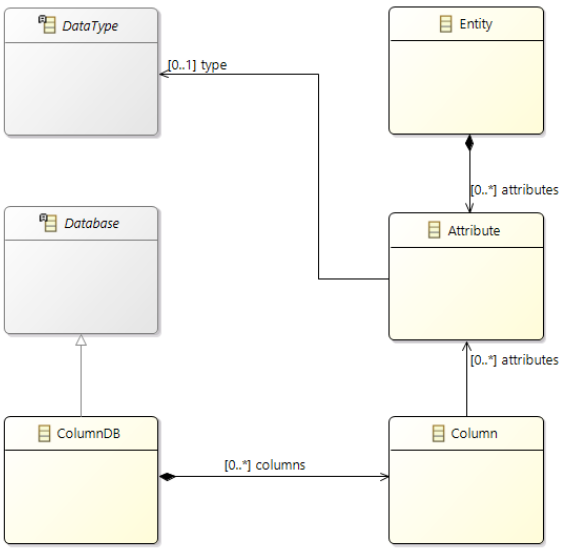

Figure 10: Column database concepts

ColumnDB Column databases store groups of columns for all rows as the basic storage unit. Thus, the *ColumnDB* metaclass consists of the reference *columns*, which permits to create *Column* elements defined below.

Column It permits to specify the attributes (defined in the conceptual entities) that should be stored in the column being modeled.

3.6 TyphonML change operators

According to the collaboration established with WP6, the TyphonML language has been refined in order to provide modelers with constructs that are necessary to specify changes (directly from the modeling tools) to be operated on existing TyphonML specifications. The changes that the migration tools being developed in WP6 are able to automatically manage are those shown in Table 4. It is important to remark that this document discusses the available change operators from a syntactical point of view. The semantics of the available operators in terms of the actual actions that are executed to perform the specified changes is detailed in the deliverable D6.2 [4].

To support the change operators in Table 4 the TyphonML has been refined by adding the metamodeling elements, which extend the new *ChangeOperator* metaclass shown in Fig. 4 as described in the following.

3.6.1 Change operators for conceptual elements

Figure 11 shows the metaclasses related to the operators that are available for changing existing TyphonML specifications with respect to conceptual entities. In particular, the language permits modelers to add new entities (see the *AddEntity* metaclass), remove existing ones (*RemoveEntity*), and even rename them (*RenameEntity*). Modelers have also the possibility to split (horizontally or vertically [4]) an existing entity (*SplitEntity*)

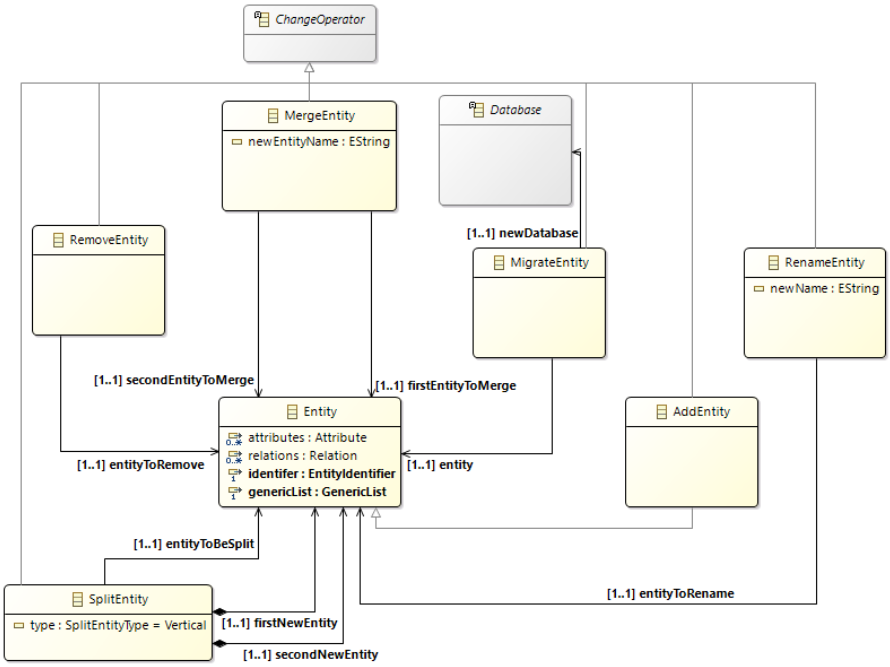

Figure 11: Operators for specifying *Entity* changes

TyphonML element	Level	Change operator
Entity	Conceptual	Add Remove Rename Split Migrate Merge
Relationship	Conceptual	Add Remove Rename Enable containment Disable containment Enable bidirectional relationship Disable bidirectional relationship Change cardinality
Attribute	Conceptual	Add Change type Remove Rename
Table	Logical/Database	Rename
Identifier	Logical/Database	Add Add component Remove Remove component Rename
Index	Logical/Database	Add Remove Add component Remove component
Collection	Logical/Database	Rename Add Index Drop Index

Table 4: Supported change operators (from D6.2 [4])

or merge two of them in a new one (*MergeEntity*). TyphonML permits also to move a conceptual entity from a given database type to another one (*MigrateEntity*).

Figure 12 shows the metaclasses related to the operators that are available for changing existing TyphonML specifications with respect to conceptual relationships. According to the metamodel fragment shown in Fig. 12, relationships can be added (*AddRelation*), removed (*RemoveRelation*), renamed (*RenameRelation*), made bidirectional or unidirectional (*ChangeRelationBidirectionality*). Moreover, it is also possible to change their cardinality (*ChangeRelationCardinality*) and specify different containment prescriptions (*ChangeRelationContainment*).

Figure 13 shows the metaclasses related to the operators that are available for changing existing TyphonML specifications with respect to attributes of given conceptual entities. According to the metamodel fragment shown in Fig. 13, attributes can be added to entities (*AddAttribute*), removed (*RemoveAttribute*), renamed (*RenameAttribute*), and also can change their type (*ChangeAttributeType*).

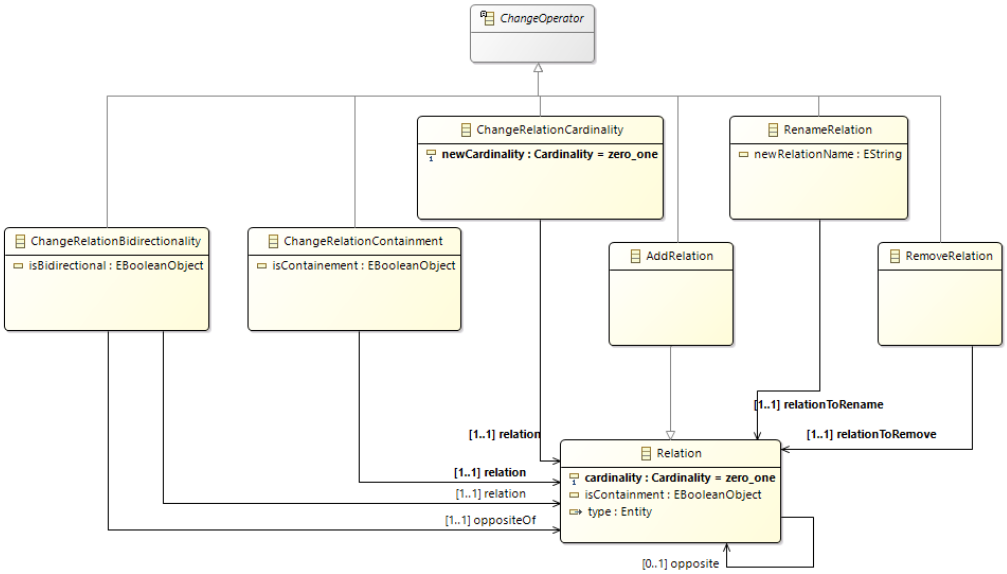

Figure 12: Operators for specifying *Relationship* changes

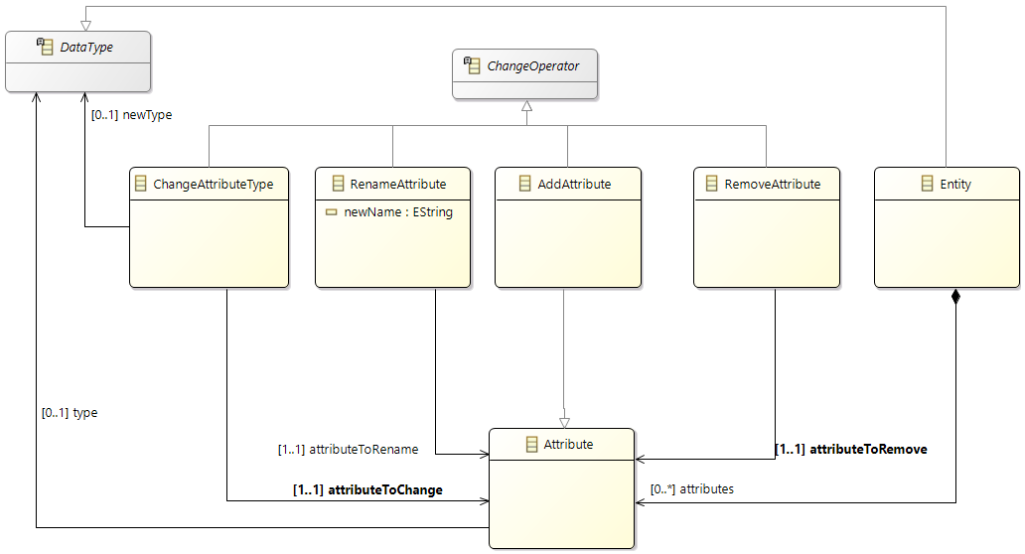

Figure 13: Operators for specifying *Attribute* changes

3.6.2 Change operators for logical elements

TyphonML enables also the specification of changes to be operated at logical level. Figure 14 shows modifications that modelers might want to operate on tables in case of relational databases. In particular, it is possible to rename existing tables (see the metaclass *RenameTable* in Fig. 14) and also specify changes on corresponding table identifiers and indexes. In particular, modelers can add or remove entity identifiers (see *AddIdentifier* and *RemoveIdentifier*, respectively), further than changing the composition of the existing identifiers in terms of the constituting attributes (see *AddAttributesToIdentifier*, and *RemoveAttributesToIdentifier*). Moreover, modelers can add and remove table indexes (see *AddIndex* and *DropIndex*, respectively), further than changing

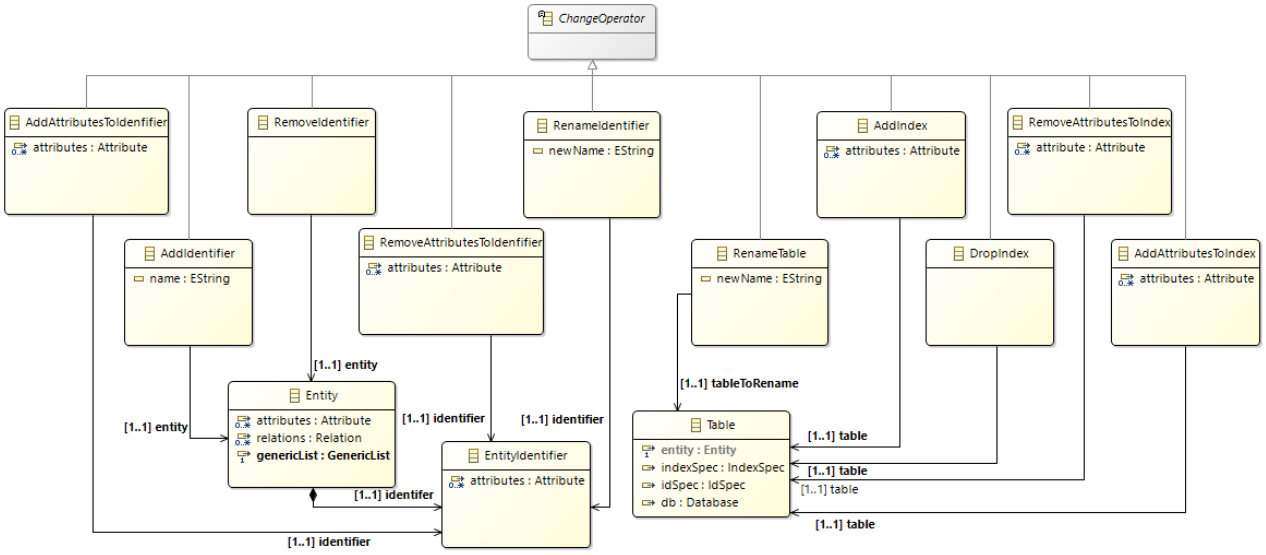

Figure 14: Operators for specifying logical changes involving *Table*, *Identifier*, and *Index* elements

the attributes that are considered for indexing the contents of the table being modified (see the metaclassess *RemoveAttributesToIndex* and *AddAttributesToIndex*).

Figure 15 shows the operator changes to be operated at logical level on database systems based on collections. In particular, the changes that are supported are collection renaming (*RenameCollection*), and addition (*AddCollectionIndex*), and removal (*DropCollectionIndex*) of indexes for the collection being modified.

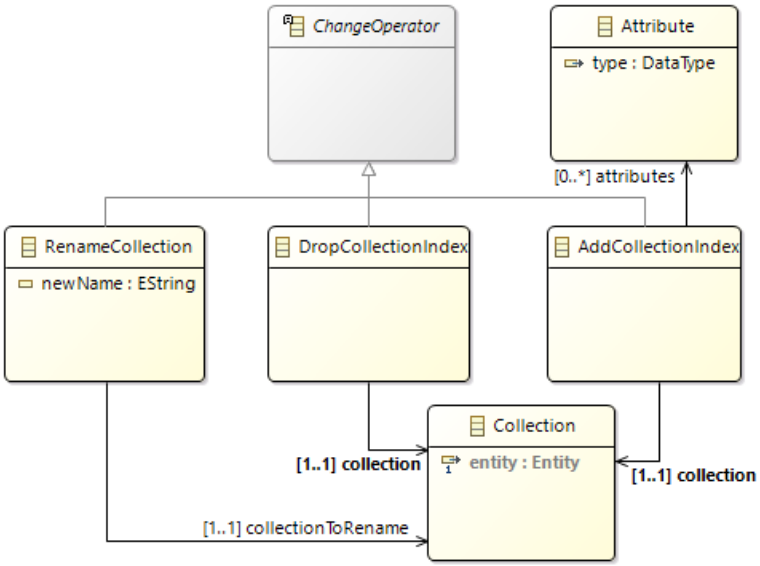

Figure 15: Operators for specifying logical changes involving *Collection* elements

3.6.3 Use of the TyphonML change operators

It is important to remark that the change operators presented in the previous section are intended to support evolution and migration scenarios as detailed in the deliverable D6.2 [4]. In particular, as shown in Fig. 16, starting from an existing and already deployed *TyphonML model*, modelers might have the need of operating conceptual and logical changes to the managed data. To this end, modelers can specify the needed changes by exploiting an *evolution editing mode* being provided by the TyphonML modeling editors (that are planned to be released at M18). In such a mode, only sequential applications of the operators described in the previous section are allowed in terms of *Evolution Scripts* as shown in Fig. 16. Such sequences are consumed by the evolution tools being developed in WP6 in order to perform all the tasks that are needed to evolve and migrate the existing polystore and bring it to a new state, which is consistent with respect to the new TyphonML specification (see *Evolved TyphonML model* in Fig. 16).

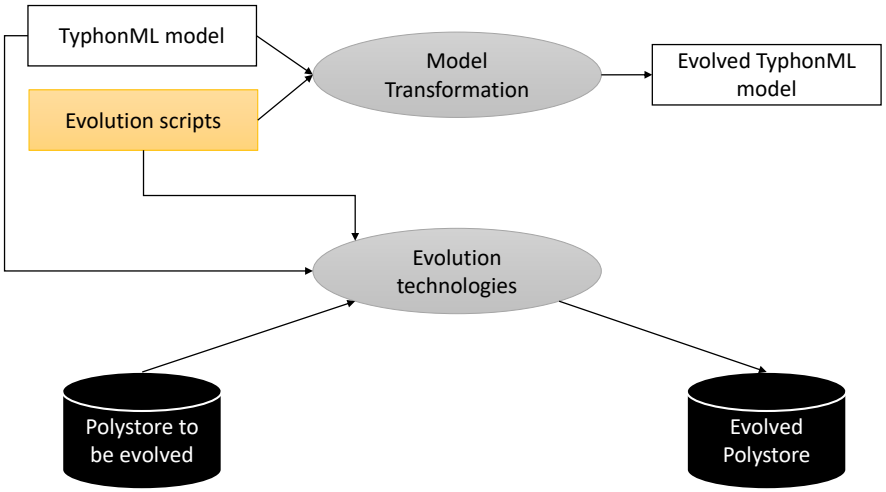

Figure 16: Use of the TyphonML change operators

3.7 Enabling natural language processing

In order to enable advanced text analysis, Edge Hill University (EHU) is working on natural language processing tasks that can be applied on polystores, which are modeled by means of TyphonML.

As detailed in D2.2 [5], different text processing tasks have been elicited by EHU; the enumeration *NlpTaskType* shown on the left end side of Fig. 17 has been defined so to enable the application of all the text processing tasks elicited so far by EHU. Such tasks are enabled on all the conceptual attributes that are typed by modelers as *FreeText*. Details about the analysis and the management of *FreeText* attributes are given in D2.2. For the sake of this document, it is enough to mention that an Elasticsearch[1] back-end is configured to enable the application of natural language processing tasks as specified in the TyphonML models.

[1] https://www.elastic.co/

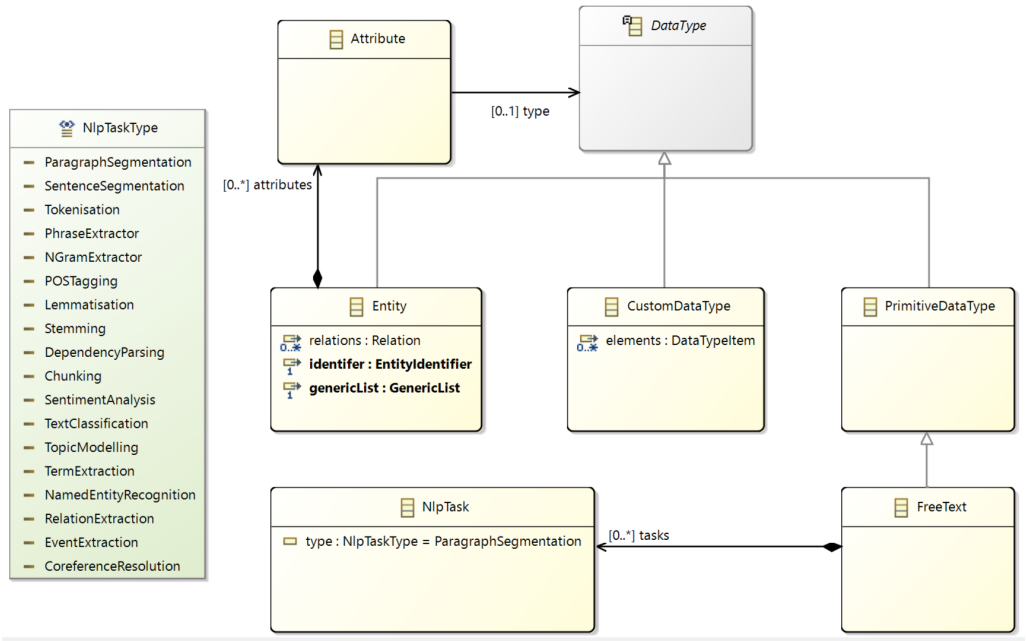

Figure 17: *Freetext* data type and corresponding natural language processing tasks

4 Hybrid Polystore Access Infrastructure and API

The metamodel presented in the previous section has been implemented as an EMF/Ecore [1] model. An initial version of the Eclipse-based TyphonML textual editor has been also developed by means of Xtext[2]. We plan to implement and release at M18 also a graphical editor in order to satisfy the requirements of the TYPHON industrial partners that have explicitly expressed the needs for both textual and graphical editors.

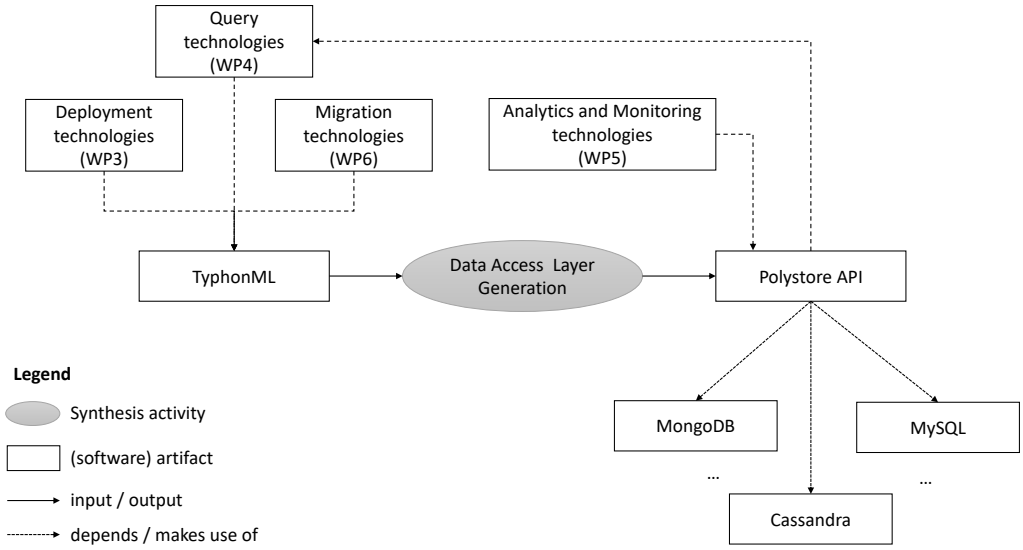

Figure 18: Use of TyphonML specifications

Further than underpinning the implementation of textual and graphical editors, the TyphonML metamodel plays a key role also for developing additional supporting tools including the software able to generate ready-to-use APIs (see Figure 18) through which application developers are able to perform CRUD (create, read, update and delete) operations and queries on the modeled hybrid polystores.

This section presents the details about the *Data Access Layer Generation* tools that have been developed to automatically generate a *Polystore API* out of an input TyphonML model.

4.1 A microservice-based architecture for the Polystore API

Lewis and Fowler define the microservice architectural style as an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API [2]. Microservices allow large systems to be built up from many collaborating components. These services are built around business capabilities and independently deployable by fully automated deployment machineries. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

By adhering to such microservice principles, we conceived the architecture model shown in Fig. 19 to manage any polystore, which is modeled by means of TyphonML. In particular, each modeled database system induces the creation of a corresponding microservice, which is responsible of managing all the conceptual entities that have been assigned to that database system. The architecture consists of a *Client Library* that is a library that

[2] `https://www.eclipse.org/Xtext/`

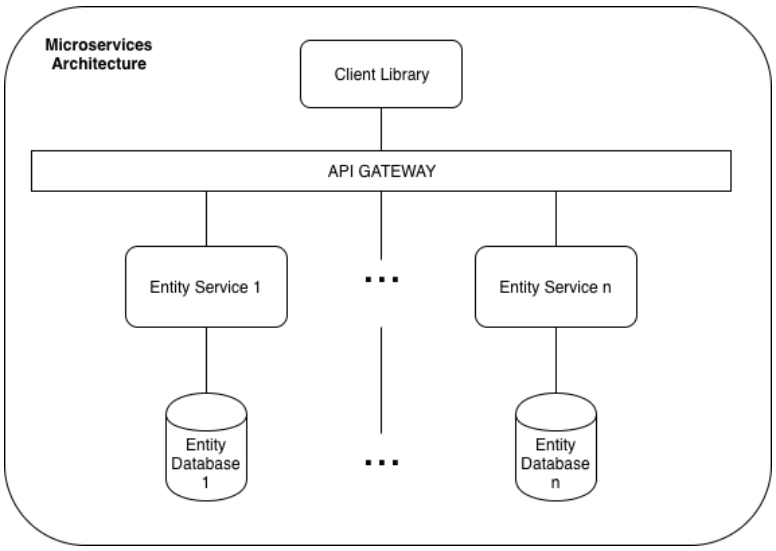

Figure 19: Proposed microservice-based architecture for the Polystore API

can be used by the developers and interacts with the interfaces exposed by the *API Gateway*. The *API Gateway* is the service that knows where all the services managing the conceptual entities are deployed, and thus it is aware of where the client requests have to be forwarded to. The system is also able to manage relationships occurring among entities stored in different database systems, and thus managed by different microservices.

It is important to remark that the actual implementation of the architecture shown in Fig. 19 relies on the Spring Framework[3]. In particular, the following technologies are used:

- **Spring Boot**[4]: it makes easy to create stand-alone, production-grade Spring based Applications;
- **Spring Data**[5]: it is a project that contains many subprojects that are specific to a given database. It defines a Spring-based programming model for data access while still retaining the special traits of the underlying data store and it makes easy to use different data access technologies e.g., relational and non-relational databases, map-reduce frameworks, and cloud-based data services.
- **Spring Cloud**[6]: it provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state).

Spring Data plays a key role in the proposed approach since it is used to actually access data. Such a layer can be potentially replaced by or used in conjunction with the access layer provided by WP4 once available, without disrupting the generation of the proposed polystore access infrastructure.

As shown in Fig. 18, the Polystore API that reflects the architecture shown in Fig. 19 is automatically generated from the given TyphonML model. In the next sections, details about such a data access layer generation are given.

[3] `https://spring.io/`
[4] `https://spring.io/projects/spring-boot`
[5] `https://spring.io/projects/spring-data`
[6] `https://spring.io/projects/spring-cloud`

4.2 The Data Access Layer Generation Process

The generation of the data access layer out of an input TyphonML model has been developed as a set of coordinated model-to-code transformations. To this end, Acceleo [7] has been adopted. It is an open-source code project belonging to the Eclipse ecosystem [8] that allows developers to employ model-driven principles to build applications. An Acceleo based generator consists of several modules, each specified in terms of templates (that are sets of Acceleo statements used to generate text) and/or queries (to extract information from the manipulated models).

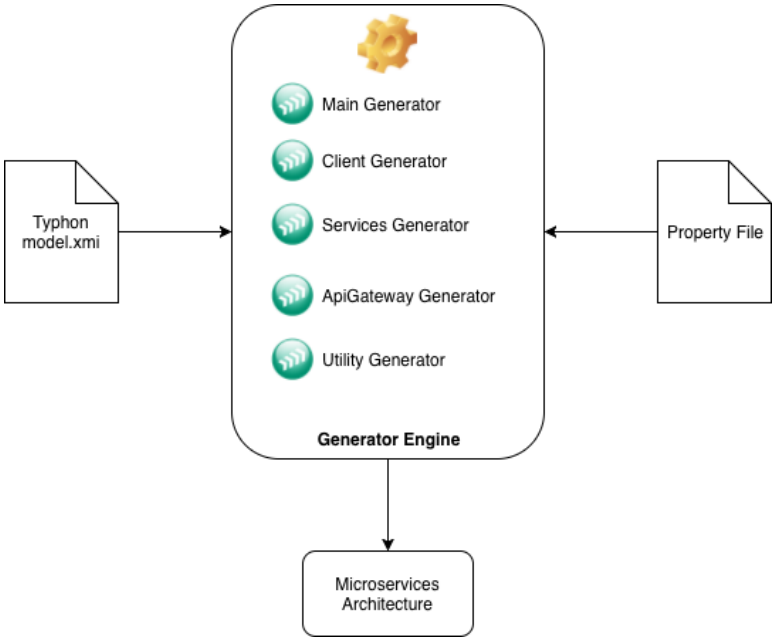

Figure 20: Generation Process.

The developed generator[9] is shown in Fig.20 and consists of the following Acceleo modules:

- *mainGenerator.mtl*: it mainly orchestrates the execution of the other modules;
- *clientGenerator.mtl*: it deals with the generation of the *Client Library* service;
- *serviceGenerator.mtl*: it deals with the generation of all the entity services with all the information needed by the Spring Data framework to initialize the proper databases);
- *apigatewayGenerator.mtl*: it deals with the generation of the *Api Gateway* service;
- *utilityGenerator.mtl*: it produces utility functions that are exploited at run-time by the generated system.

In the following, the different Acceleo modules are described with more details. In particular, the TyphonML model on the left-hand side of Fig. 21 representing the conceptual entities shown in Fig. 22. is taken as input, and the target application shown in the lower hand side of Fig. 21 is generated.

Main Generator As shown in the code fragment in Fig. 23, the main generator module has the role of executing all the other modules by applying them on the input TyphonML model.

[7] https://www.eclipse.org/acceleo/
[8] https://www.eclipse.org/
[9] https://github.com/typhon-project/typhonml/tree/master/it.univaq.disim.typhon.acceleo

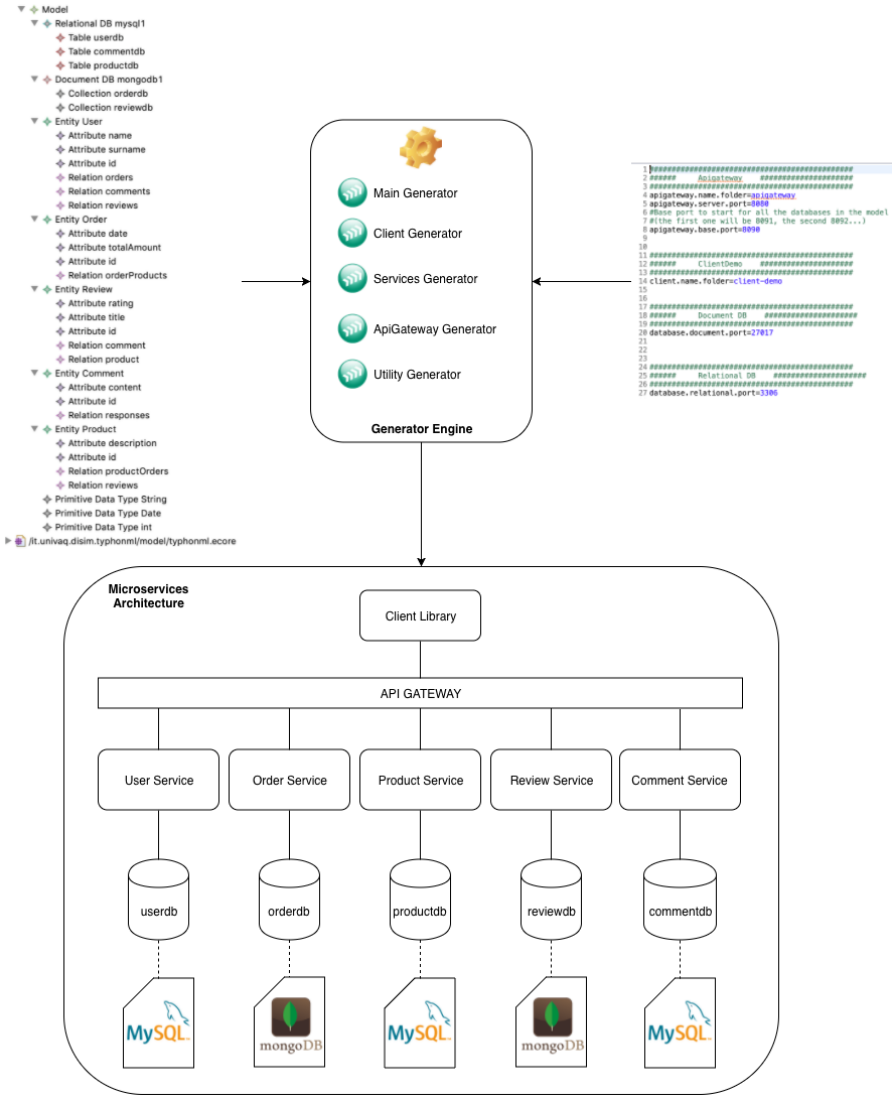

Figure 21: Sample application of the proposed generation process

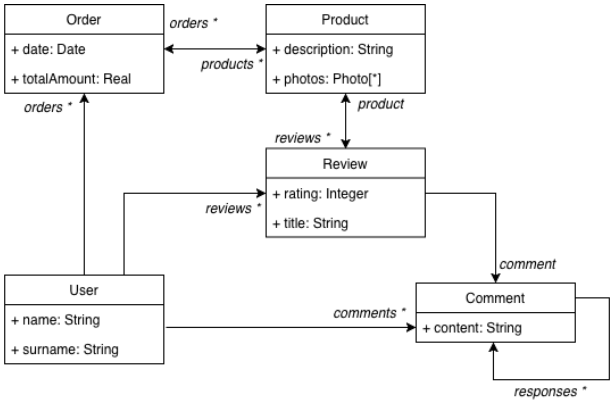

Figure 22: Data entities of an explanatory e-commerce system

```
[comment encoding = UTF-8 /]
[module generate('http://org.typhon.dsls.typhonml.sirius')]

[import it::univaq::disim::typhon::acceleo::files::apigatewayGenerator /]
[import it::univaq::disim::typhon::acceleo::files::clientGenerator /]
[import it::univaq::disim::typhon::acceleo::files::servicesGenerator /]


[template public generateAll(aModel : Model)]
[comment @main/]

[apigatewayGenerator(aModel)/]
[clientGenerator(aModel)/]
[servicesGenerator(aModel)/]

[/template]
```

Figure 23: Fragment of the *Main Generator* module

Client Generator This module is demanded to create all the files needed to build the *Client Library* (see Fig.19) that will be used by developers to perform their operations over the microservices architecture. Essentially it cycles over all the model entities (see Fig.24.a) in order to build a Java project with all the services for all the entities in the source model (see Fig.24.b). A fragment of the source code implementing the service managing *Order* elements is shown in Fig. 24.c.

Figure 24: *a)* Fragment of the *Client Generator b)* Structure of the generated *Client Library* project *c)* Fragment of the generated *OrderService*

Services Generator This module creates as many services as the entities modelled in the TyphonML model. For each service a corresponding Java project is generated (see Fig. 25). For instance, by considering the online shop model shown in Fig. 22, the *Order* has been assigned to a document database, whereas the *Product* is assigned to a relational database. Thus, the *Service Generator* adds the right annotation, which is then properly consumed by the Spring Data framework as shown in the generated *Order* and *Product* Java classes as shown in Fig.26.a and Fig.26.b, respectively.

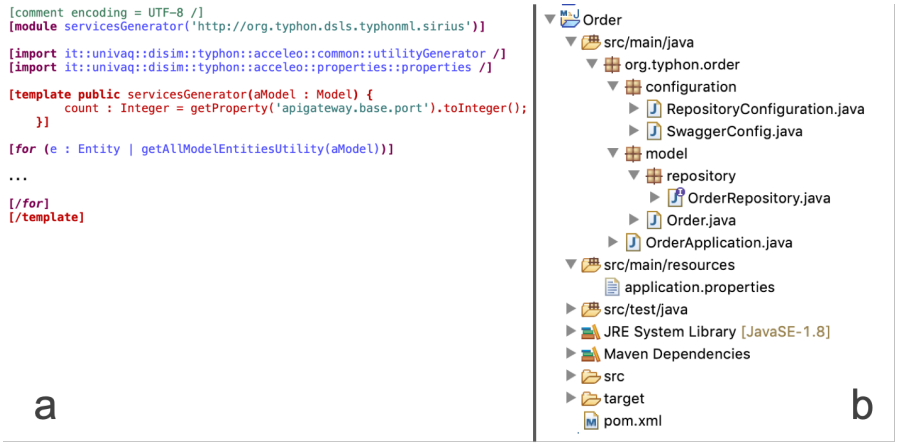

Figure 25: *a)* Fragment of the *Service Generator* module *b)* A generated service project

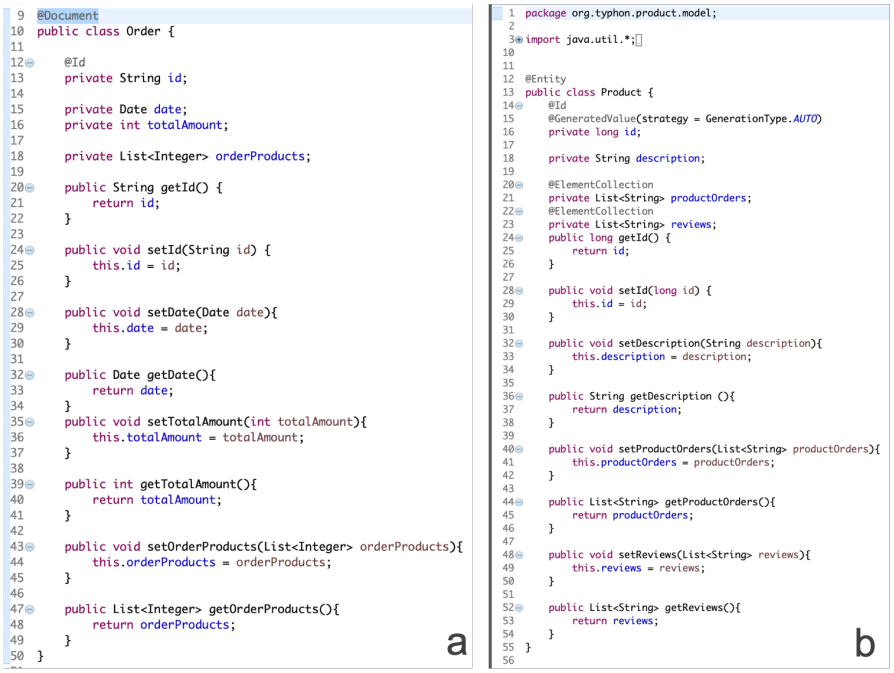

Figure 26: *a)* Document database Spring Data Annotation example (Order Entity) *b)* Relational database Spring Data Annotation example (Product Entity)

API Gateway Generator The critical aspect of the *API Gateway* is its ability to forward to the right services the requests coming from the *Client Library*. To this end, the API gateway needs to know all the available services, the ports they are listening to, etc. To this end, the API gateway generator iterates on all the conceptual entities and properly assigns them a progressive number in which the related service is listening (see Fig.27). This port number will be the same port that is assigned to each service by the *Service Generator*.

```
[comment encoding = UTF-8 /]
[module apigatewayGenerator('http://org.typhon.dsls.typhonml.sirius')]

[import it::univaq::disim::typhon::acceleo::common::utilityGenerator /]


[template public apigatewayGenerator(aModel : Model)
    {
        basePort : Integer = getProperty('apigateway.base.port').toInteger();
    }]


[comment org.typhon.apigateway.ApigatewayApplication.java/]
[file (getProperty('apigateway.name.folder')+'/src/main/java/org/typhon/apigateway/ApigatewayApplication.java', false, 'UTF-8')]
package org.typhon.apigateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.gateway.route.RouteLocator;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

//import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@SpringBootApplication
//@EnableSwagger2
public class ApigatewayApplication {

    @Bean
    RouteLocator gatewayArtifactServiceRouter(RouteLocatorBuilder builder){
        return builder.routes()
            [for (e : Entity | getAllModelEntitiesUtility(aModel))separator('\n')]
                .route(r -> r.path("/[e.name.toLowerCase()/]/**").and().method("GET")
                        .filters(f-> f.rewritePath("/[e.name.toLowerCase()/]","/[e.name.toLowerCase()/]"))
                        .uri("http://localhost:[basePort + aModel.dataTypes->filter(typhonml::Entity)->sortedBy(x | x.name)->indexOf(e)/]"))
                .route(r -> r.path("/[e.name.toLowerCase()/]").and().method("POST")
                        .filters(f-> f.rewritePath("/[e.name.toLowerCase()/]","/[e.name.toLowerCase()/]"))
                        .uri("http://localhost:[basePort + aModel.dataTypes->filter(typhonml::Entity)->sortedBy(x | x.name)->indexOf(e)/]"))
                .route(r -> r.path("/[e.name.toLowerCase()/]/**").and().method("PUT")
                        .filters(f-> f.rewritePath("/[e.name.toLowerCase()/]","/[e.name.toLowerCase()/]"))
                        .uri("http://localhost:[basePort + aModel.dataTypes->filter(typhonml::Entity)->sortedBy(x | x.name)->indexOf(e)/]"))
                .route(r -> r.path("/[e.name.toLowerCase()/]/**").and().method("DELETE")
                        .filters(f-> f.rewritePath("/[e.name.toLowerCase()/]","/[e.name.toLowerCase()/]"))
                        .uri("http://localhost:[basePort + aModel.dataTypes->filter(typhonml::Entity)->sortedBy(x | x.name)->indexOf(e)/]"))
            [/for]
                .build();
    }

    public static void main(String[ '[' /][ ']' /] args) {
        SpringApplication.run(ApigatewayApplication.class, args);
    }
}
[/file]
```

Figure 27: Fragment of the *API Gateway Generator* module

Utility Generator The *Utility Generator* module generates all the utility templates and queries that are used during the generation processes. For example, concerning the services mapped in the *API Gateway* and the actual services created by the *Service Generator*, it is essential that they get the same port number (so that the can communicate with each other), and to do so the way to collect the entities from the model have to be consistent. So it has been centralized in this utility module the query that does that. For instance, Fig.28 shows a query that through an OCL construct retrieves all the entities and sort them based on their name.

4.3 Managing data relationships

One of the critical aspects that have been taken into account when developing the polystore API generator has been the management of the data relationships linking entities that are stored in different databases, and thus that are managed by different microservices. To this end, additional attributes are added in the logical schemas

```
[comment encoding = UTF-8 /]
[module utilityGenerator('http://org.typhon.dsls.typhonml.sirius')]

[query public getAllModelEntitiesUtility(aModel : Model) : Entity = aModel.dataTypes->filter(typhonml::Entity)->sortedBy(x | x.name) /]

[comment
################################################################################
##### START - POLYMORFISM FOR getting the name of database starting from its entity
################################################################################
/]
[template public resolveEntityNameFromDatabase(database : Database, entity : Entity, aModel : Model) ]
    [comment TODO Auto-generated template stub/]
[/template]
[template public resolveEntityNameFromDatabase(database : RelationalDB, entity : Entity, aModel : Model) ]
[for (table : Table | aModel.eAllContents()->filter(typhonml::RelationalDB).tables)][if (table.entity.name.strcmp(entity.name) = 0)][table.name/][/if][/for]
[/template]
[template public resolveEntityNameFromDatabase(database : DocumentDB, entity : Entity, aModel : Model) ]
[for (collection : Collection | aModel.eAllContents()->filter(typhonml::DocumentDB).collections)][if (collection.entity.name.strcmp(entity.name) = 0)][collection.name/][/if]
[/for]
[/template]
[template public resolveEntityNameFromDatabase(database : KeyValueDB, entity : Entity, aModel : Model) ]
[for (element : KeyValueElement | aModel.eAllContents()->filter(typhonml::KeyValueDB).elements)][if (element.entity.name.strcmp(entity.name) = 0)][element.key/][/if][/for]
[/template]
[template public resolveEntityNameFromDatabase(database : GraphDB, entity : Entity, aModel : Model) ]
    [comment TODO Auto-generated template stub/]
[/template]
[template public resolveEntityNameFromDatabase(database : ColumnDB, entity : Entity, aModel : Model) ]
    [comment TODO Auto-generated template stub/]
[/template]
[comment
################################################################################
###### END - POLYMORFISM FOR getting the name of database starting from its entity
################################################################################
/]


[template public calculateEntityPortByRelation(aModel : Model, relation : Relation){
        basePort : Integer = getProperty('apigateway.base.port').toInteger();
    }]
[for (ent : Entity | getAllModelEntitiesUtility(aModel))][if(ent.name.strcmp(relation.type.name) = 0)][basePort + aModel.dataTypes->filter(typhonml::Entity)->sortedBy(x | x.name)->indexOf(ent)/][/if][/for]
[/template]

[template public calculateEntityPort(aModel : Model, entity : Entity){
        basePort : Integer = getProperty('apigateway.base.port').toInteger();
    }]
[for (ent : Entity | getAllModelEntitiesUtility(aModel))][if(ent.name.strcmp(entity.name) = 0)][basePort + aModel.dataTypes->filter(typhonml::Entity)->sortedBy(x | x.name)->indexOf(ent)/][/if][/for]
[/template]

```

Figure 28: Fragment of the *Utility Generator* module

managing the related entities by distinguishing *one-to-many* and *many-to-many* relationships as described in the following.

4.3.1 One-to-Many relationships

An explanatory example of one-to-many relationship is shown in Fig.22 between the *User* and *Order* entities (i.e., a user can have several orders). According to the input TyphonML model, the generator described in the previous section generates one microservice for managing *User* objects (that are stored in a MySQL database), and one microservice for managing *Order* objects (that are stored in a MongoDB database). As shown in Fig.29, the *userdb* contains a table *user_orders* to manage the relations among user and order objects. Orders are referenced by means of strings representing order IDs.

The management of such a relation is performed by the *Client Library* service in a transparent manner for the developer. For instance, the *User.java* file generated because of the *User* entity contains the *getOrderObj()* method (see Fig. 30), which deals with the recovery of referenced data in a lazy loading[10] manner.

4.3.2 Many-to-Many relationships

Many-to-Many relationships are managed in a manner, which is similar to the one-to-many case. An example of many-to-many relationship is shown in Fig.22 between the *Order* and *Product* entities. The latter is managed by a MySQL database. In such cases, the generator add specific elements in the logical schemas to enable the retrieval of linked objects. For instance, as shown on the left-hand side of Fig. 31, the *order* collections contain the array *orderProducts* to refer the *Product* elements that are contined in the considered orders. Moreover, as shown on the right-hand side of the same figure, the *productdb* database constains the table *product_product_orders* to manage the relationship between product and order objects. Such details are completely hidden to the developers, who are provided with the *Client Library* that allows to navigate relationships by means of dedicated methods (see Fig. 32) in the corresponding generated *Product.java* and *Order.java* files (see Fig.24.b).

[10] https://en.wikipedia.org/wiki/Lazy_loading

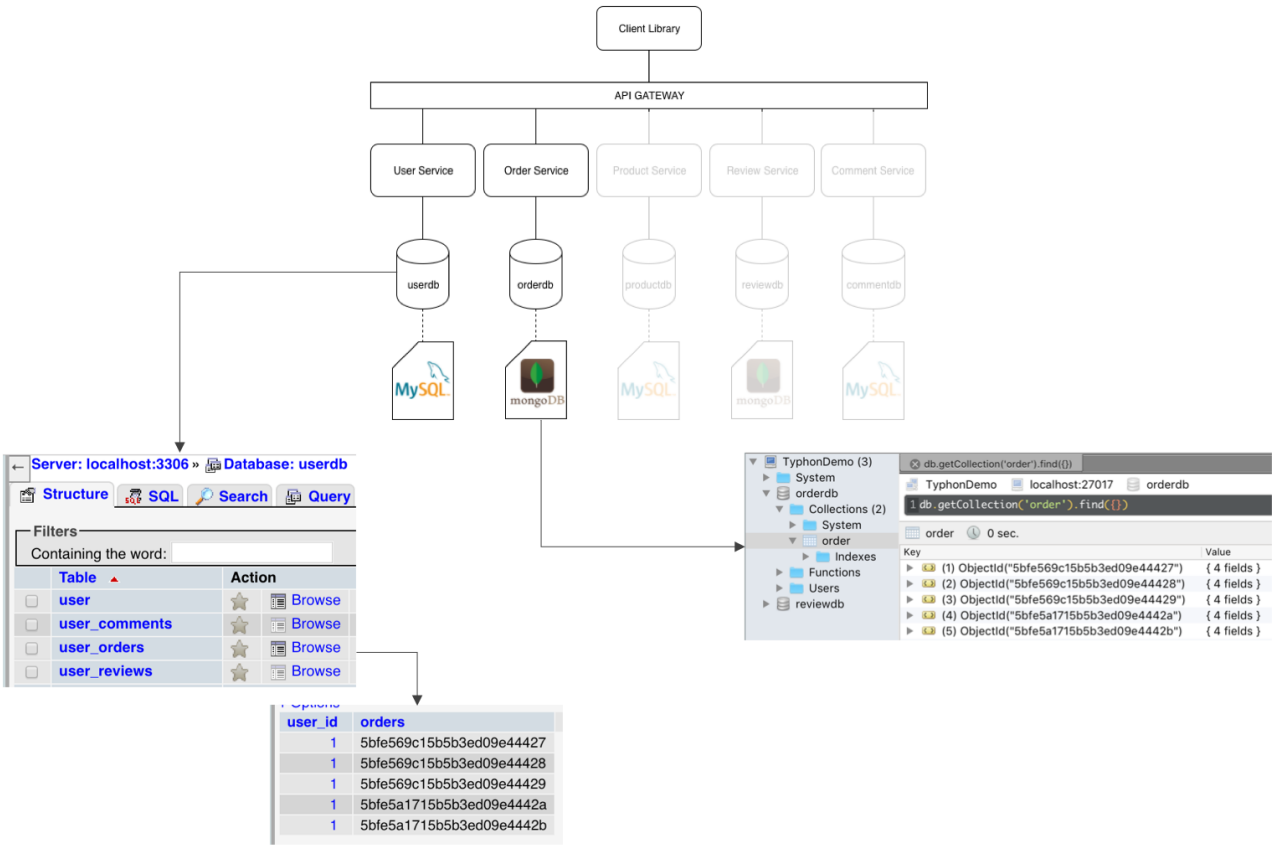

Figure 29: One-to-Many mapping example

```
public List<Order> getOrderObj() {
    OrderService orderService = new OrderService("http://localhost:8092");
    List<Order> result = new ArrayList<Order>();
    for (String typeObj : orders) {
        try {
            result.add(orderService.findById(typeObj));
        }
        catch (Exception e) {
            logger.error(e.getMessage());
        }
    }
    return result;
}
```

Figure 30: Method of the *User* class retrieving related *Order* objects

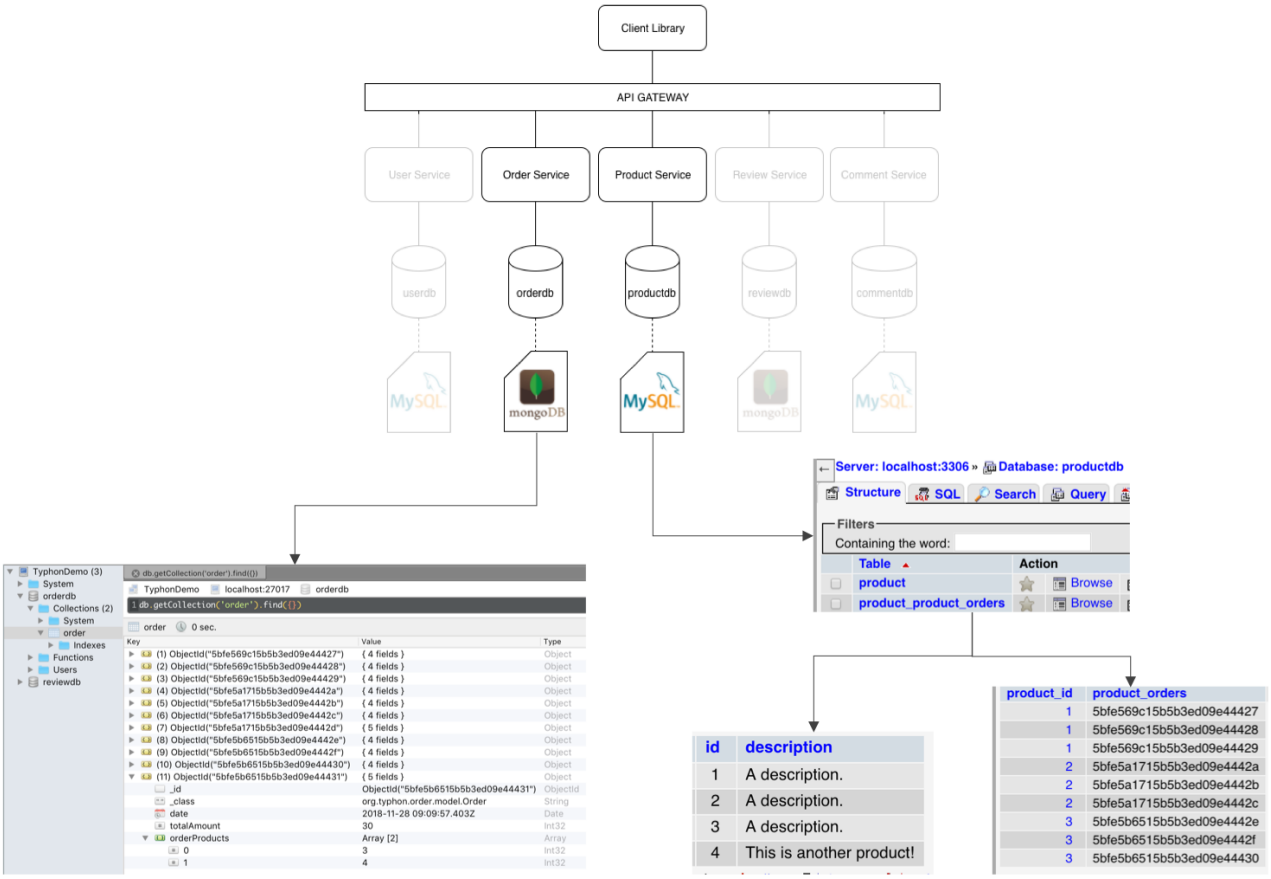

Figure 31: Many-to-Many Mapping Example

```
public List<Product> getProductObj() {
    ProductService productService = new ProductService("http://localhost:8093");
    List<Product> result = new ArrayList<Product>();
    for (Integer typeObj : orderProducts) {
        try {
            result.add(productService.findById(typeObj));
        }
        catch (Exception e) {
            logger.error(e.getMessage());
        }
    }
    return result;
}
```

a

```
public List<Order> getOrderObj() {
    OrderService orderService = new OrderService("http://localhost:8092");
    List<Order> result = new ArrayList<Order>();
    for (String typeObj : productOrders) {
        try {
            result.add(orderService.findById(typeObj));
        }
        catch (Exception e) {
            logger.error(e.getMessage());
        }
    }
    return result;
}
```

b

Figure 32: *a)* Fragment of the *getProductObj* in the generated *Order* class *b)* Fragment of the *getOrderObj()* in the generated *Product* class

5 Conclusions

In this document we presented the results of WP2 related to the design and development of the TyphonML language aiming at supporting the design and development of hybrid polystores. The changes performed on the initial version of the language as presented in the deliverable D2.1.

The expressiveness of the language has been assessed *i)* by considering the WP2 and use case requirements elicited during the first six months of the project and detailed in the deliverable D1.1, and *ii)* by developing the tools that are able to automatically generate the data access layer infrastructure out of source TyphonML specifications.

It is important to recall that the TyphonML model will underpin the development of the corresponding modeling tools providing developers with both textual and grapical editors. Thus, we expect some minor refinements of the TyphonML metamodel also to meet the technical requirements of the tools being developed in the other workpackages.

References

[1] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T.J. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.

[2] M. Fowler and J. Lewis. Microservices a definition of this new architectural term. url: `http://martinfowler.com/articles/microservices.html`.

[3] The University of L'Aquila. D2.1 – Hybrid Polystore Modelling Language (Interim Version), 2018.

[4] The University of Namur. D6.2 – Hybrid Polystore Schema Evolution Methodology and Tools, 2018.

[5] The Edge Hill University. D2.2 – Text Modelling Extension, 2018.

[6] The Open Group with contributions from all partners. D1.1 - Project Requirements, 2018.