Project Number 780251

D6.5 Hybrid Polystore Continuous Evolution Tools

Version 1.0
8 July 2020
Final

Public Distribution

University of Namur

Project Partners: **Alpha Bank**, **ATB**, **Centrum Wiskunde & Informatica**, **CLMS**, **Edge Hill University**, **GMV**, **OTE**, **SWAT.Engineering**, **The Open Group**, **University of L′Aquila**, **University of Namur**, **University of York**, **Volkswagen**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the TYPHON Project Partners accept no liability for any error or omission in the same.

© 2020 Copyright in this document remains vested in the TYPHON Project Partners.

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
OTE Theodoros E. Mavroeidakos 99 Kifissias Avenue 151 24 Athens Greece Tel: +30 697 814 7618 E-mail: tmavroeid@ote.gr	**SWAT.Engineering** Davy Landman Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 633754110 E-mail: davy.landman@swat.engineering
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	19 March 2020
0.4	First draft for internal review	17 June 2020
0.9	Full draft for partner review	3 July 2020
1.0	Updates from QA Review	8 July 2020

Table of Contents

1 Introduction **1**

1.1 Purpose of the deliverable . . . 1

1.2 Relationship to other Typhon deliverables . . . 1

1.3 Contributors . . . 2

1.4 Structure of the deliverable . . . 2

2 Continuous evolution tool **3**

2.1 Step 1: Capturing TyphonQL queries . . . 4

2.2 Step 2: Parsing and classifying TyphonQL queries . . . 4

2.3 Step 3: Visual analytics of polystore data usage . . . 7

2.4 Step 4: Recommending polystore schema reconfigurations . . . 17

2.5 Step 5: Applying the selected recommendations . . . 18

3 Data Ingestion Tool **21**

3.1 Step 1: Extraction . . . 21

3.2 Step 2: Deployment . . . 25

3.3 Step 3: Ingestion . . . 25

4 Summary of WP6 contributions **26**

5 Conclusions **30**

List of Figures

1 General architecture of the continuous evolution tool . . . 3

2 TyphonML model of the polystore used as illustrative example . . . 5

3 Structure of the analytics database (MongoDB) . . . 6

4 Main view of the continuous evolution tool . . . 8

5 Global schema view of the polystore . . . 8

6 Schema view of the relational database . . . 9

7 Schema view of the document database . . . 9

8 Entity size view . . . 10

9 Size of the (selected) polystore entities over time, expressed in number of records . . . 11

10 Distribution of CRUD operations applied to the polystore . . . 11

11 Distribution of queries among the polystore entities . . . 12

12 Distribution of CRUD operations and entity accesses during a particular period of time 13
13 Distribution of CRUD operations over time, expressed in number of queries 14
14 Number of queries accessing the (selected) polystore entities over time 14
15 General overview of the most frequent query types (left), and of the slowest queries (right) . . 15
16 Duration of a particular query type over time 15
17 Evolution of the size of entity Address over time 16
18 Evolution of the size of entity User over time 17
19 Schema evolution recommendations related to a particular (slow) query 18
20 Selection of the schema evolution recommendations to apply to the polystore 18
21 Schema evolution operators corresponding to the recommendations selected at Figure 20 . . . 19
22 TyphonML model obtained after applying the selected recommendations 20
23 Overview of the process followed by the data ingestion tool 21
24 Example schema of an input relational database 22
25 Conceptual abstraction of input schema of Figure 24 22
26 TyphonML model extracted from the input relational schema of Figure 24 23
27 Example of configuration for the extraction step 24
28 Data ingestion parameters 25

List of Tables

1 Coverage of technical requirements related to Work Package 6 27
2 Coverage of use case requirements related to Work Package 6 28

Executive Summary

In the context of its Work Package 6, the Typhon project aims to develop a method and a technical infrastructure to support the graceful evolution of hybrid polystores, where multiple NoSQL and SQL databases may jointly 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 polystore schema to another version of a polystore schema; (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 focusses on the fourth aspect of our evolution methodology, namely the monitoring of polystore query events in order to provide users with polystore evolution recommendations, when relevant.

It also presents an additional WP6 tool, allowing one to ingest data from pre-existing relational databases to a new Typhon polystore.

1 Introduction

According to 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.

In order to reach this goal, the Typhon polystore evolution tools aim to cover 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 polystore schema to another.
- Polystore query migration: Allowing 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).

In the present deliverable, we mainly focus on the *continuous polystore evolution* aspect of our evolution methodology, the goal of which is to propose a tool supporting the continuous evolution of the hybrid polystore. This tool aims at monitoring and analyzing polystore data usage, with a particular focus on data access performance, in order to recommend polystore schema reconfigurations when relevant.

We also present an additional WP6 tool, that we developed on request of our use case partners. This tool supports the ingestion of data from pre-existing relational databases into a new polystore.

1.1 Purpose of the deliverable

This document mainly presents the work that has been done with respect to task 6.5 of Work Package 6, described as follows in the Typhon Description of Work:

Task 6.5: This task concerns the development of continuous evolution tools for the hybrid polystore. Those tools will exploit the polystore query events captured by the monitoring mechanisms developed in Work Package 5. Sophisticated algorithms will recommend possible polystore schema reconfigurations and will inform the user on the impact of recommended polystore reconfigurations.

In addition, we also present an additional tool, developed upon request of our use-case partners, which supports the ingestion of pre-existing data from relational databases into the polystore.

1.2 Relationship to other Typhon deliverables

The present deliverable is directly linked to several previous Typhon deliverables:

- The polystore continuous evolutions tools exploit the monitoring mechanisms and tools developed in Work Package 5, and presented in deliverables D5.2 [8] and D5.3 [9].
- The polystore queries subject captured and analyzed by the continuous evolution tools are expressed in the TyphonQL language developed in Work Package 4, and presented in deliverables D4.2 [1] and D4.3 [2].

- The polystore reconfigurations recommended by the continuous evolution tools are expressed as a chain of Schema Modification Operators (SMOs). Those operators are fully specified in deliverable D6.2 [5], and are integrated in the TyphonML modeling language, presented in deliverables D2.3 [3] and D2.4 [4].

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 tools presented in this deliverable. A demonstration of the data ingestion tool was presented during the Typhon project meeting in Athens (27-28 February 2020). A demonstration of the continuous evolution tools was presented during a virtual Typhon project meeting (18-19 June 2020).

1.4 Structure of the deliverable

The remainder of this Deliverable is structured as follows:

- Section 2 presents the continuous evolution tool, allowing users to monitor the data access performance of a Typhon polystore and to recommend them possible polystore reconfigurations, when relevant.
- Section 3 presents the data ingestion tool, supporting the ingestion of data from pre-existing relational databases into a new Typhon polystore.
- Section 4 summarizes the contributions of Work Package 6 and assesses the coverage of the underlying technical and use case requirements.
- Section 5 provides concluding remarks and anticipates future work in Work Package 6.

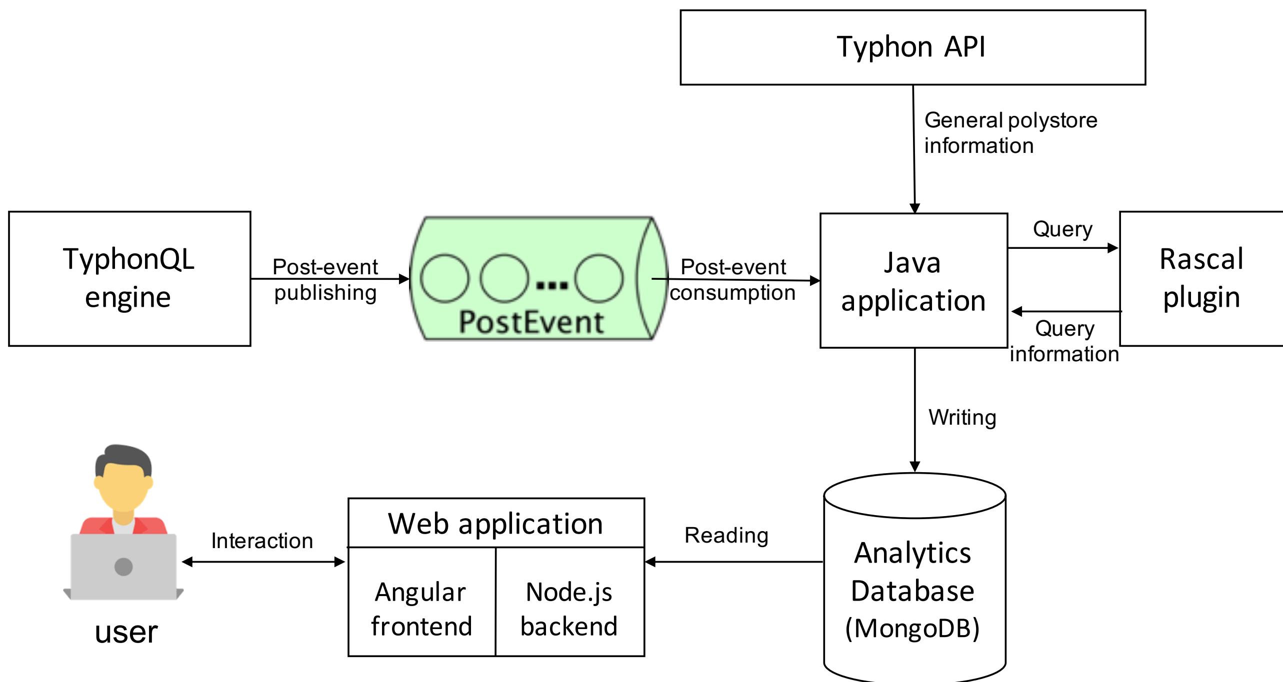

Figure 1: General architecture of the continuous evolution tool

2 Continuous evolution tool

The continuous evolution tool aims to monitor data usage performance in a Typhon polystore in order to provide users with schema evolution recommendations, when relevant.

The tool communicates with several polystore components, including the post-execution events queue (WP5) and, through the polystore API, the polystore TyphonML schema (WP2) and the polystore databases (WP3 and WP4). This allows the continuous evolution to automatically retrieve useful information about the polystore, including:

- the polystore configuration, i.e., the TyphonML entities and their mapping to underlying native databases;
- the TyphonQL queries that are executed by the TyphonQL engine, and their duration;
- the (evolving) size of the TyphonML entities over time.

General overview The general architecture of the continuous evolution tool is depicted in Figure 1. Each time a post-execution event is published to the post-event queue, a Java application wakes up and retrieves the event. If the event corresponds to a DML query execution, the Java application sends the corresponding TyphonQL query to a Rascal plugin. The latter parses, analyses and classifies the query and sends back the corresponding query information to the Java application. This information is stored in an internal MongoDB database, that is used as input by an interactive web application. The web application, relying on an Angular frontend and a Node.js backend, provides users with visual analytics of the polystore data usage, as well as with performance-based schema reconfiguration recommendations. Each of these steps is described in further details in the remaining of this section.

Running example In order to illustrate the use of the continuous evolution tool, we will use as running example a polystore structured according to the TyphonML schema given in Figure 2. This schema includes 7 conceptual entities and 2 databases. Based on this example schema, we automatically generated post-execution events corresponding to 1007 TyphonQL queries each query having a fictitious execution time. This random generation reached a more or less equal distribution in terms of CRUD operations and polystore entities (259 insert, 244 select, 243 delete, and 261 update). We also considered a fictitious history of the polystore entities, especially, as far as their size is concerned.

2.1 Step 1: Capturing TyphonQL queries

The continuous evolution tool exploits the polystore monitoring mechanisms developed in Work Package 5. Thanks to those mechanisms, the tool can capture at runtime the successive TyphonQL queries that are sent to the polystore, and executed by the TyphonQL engine. To do so, the tool consumes and analyses the so-called *post-execution events* (PostEvent), generated and pushed by the TyphonQL engine to the analytics queue of the WP5 monitoring infrastructure. We refer to deliverable D5.3 [9] for more details about this infrastructure.

2.2 Step 2: Parsing and classifying TyphonQL queries

The post-execution events captured at Step 1 include the TyphonQL queries that have been executed by the TyphonQL engine. The continuous evolution tool parses each of those queries, in order to extract relevant information to be used during the analytics and recommendation phases. Our tool focuses on post-execution events corresponding to DML queries, i.e., select, insert, delete, and update queries. It ignores other events such as, for instance, the execution of DDL queries (e.g., create entity, delete entity, etc.) sent by the schema evolution tool to the TyphonQL engine.

The tool parses each captured TyphonQL query in order to extract relevant information, including:

- the type of query (select, insert, delete, update);
- the accessed TyphonML entities;
- the join conditions, if any;
- the query execution time, expressed in ms.

The query parsing and extraction step is implemented using Rascal, based on TyphonQL syntax.

Once the query is parsed and analyzed, the tool also classifies it. This classification aims to group together all TyphonQL queries of the same form. A group of TyphonQL queries is called a *query category*. The queries belonging to the same query category are queries that would become the same query after replacing all input values with placeholders.

For instance, the following three TyphonQL queries can be classified into the same query category:

```
from Address a select a where a.country == "Belgium"
from Address a select a where a.country == "Italy"
from Address a select a where a.country == "Germany"
```

Indeed, those queries only differ in terms of their input values. When replacing the only input value corresponding to the address country ("Belgium", "Italy" and "Germany", respectively) with a placeholder ("?"), we obtain the following query category:

```
from Address a select a where a.country == "?"
```

```
entity Review{
    id : string[32]
    content : string[32]
    product -> Product[1]
}

entity Product{
    id : string[32]
    name : string[32]
    description : string[32]
    reviews :-> Review."Review.product"[0..*]
}

entity OrderProduct{
    id : string[32]
    product_date : string[32]
    totalAmount : int
    paidWith -> CreditCard[1]
}

entity User{
    id : string[32]
    name : string[32]
    comments :-> Comment[0..*]
    paymentsDetails :-> CreditCard[0..*]
    orders -> OrderProduct[0..*]
    address -> Address."Address.user"[1]
}

entity Address{
        streetName: string[32]
        streetNumber: string[32]
        zip: string[32]
        city: string[32]
        country: string[32]
        user -> User[1]
}

entity Comment{
    id : string[32]
    content : string[32]
    responses :-> Comment[0..*]
}

entity CreditCard{
    id : string[32]
    number : string[32]
    expiryDate : string[32]
}
```

```
relationaldb RelationalDatabase{
    tables{
        table {
            UserDB : User
            idSpec ('User.name')
        }

        table {
            AddressDB: Address
        }

        table {
            ProductDB : Product
            index productIndex{
                attributes ('Product.name')
            }
            idSpec ('Product.name')
        }

        table {
            CreditCardDB : CreditCard
            index creditCardIndex{
                attributes ("CreditCard.number")
            }
            idSpec ("CreditCard.number")
        }

        table {
            OrderDB : OrderProduct
            index orderIndex {
                attributes ("OrderProduct.id")
            }
            idSpec ("OrderProduct.id")
        }
    }
}

documentdb DocumentDatabase{
    collections{
            CommentsDB : Comment
            ReviewDB : Review
    }
}
```

Figure 2: TyphonML model of the polystore used as illustrative example

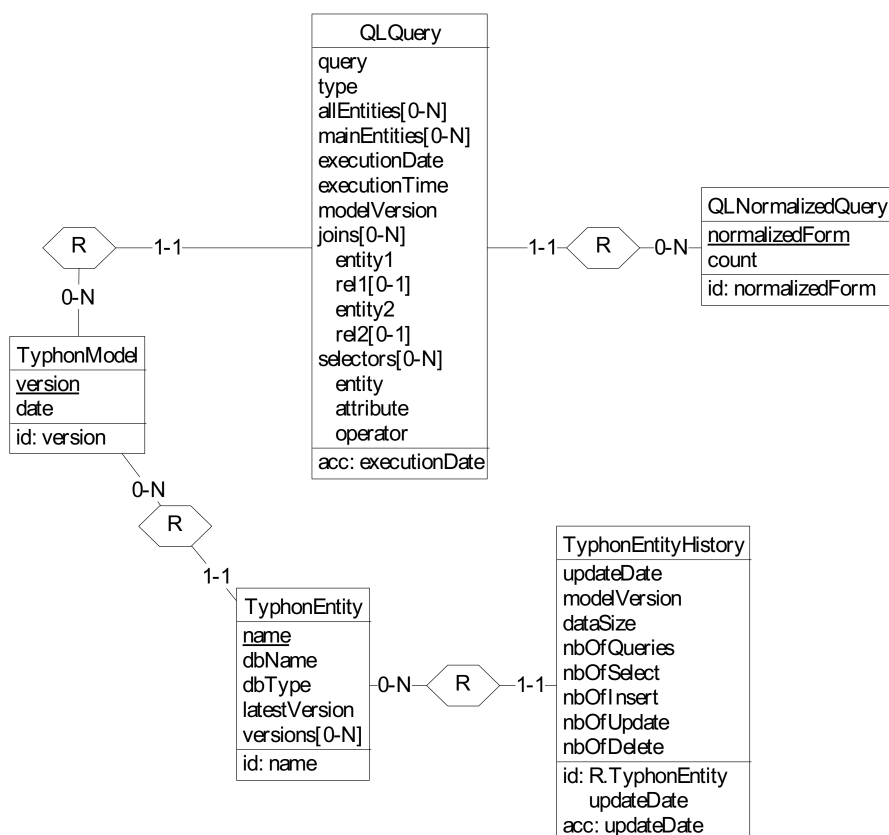

Figure 3: Structure of the analytics database (MongoDB)

In addition to parsing, analyzing and classifying the queries executed by the TyphonQL engine, the continuous evolution tool also extracts - at regular time intervals[1] - information about the Typhon polystore, with a particular focus on TyphonML entities. This includes, in particular, the size of each TyphonML entity, expressed in terms of *number of records*, e.g., number of rows for a relational table or number of documents for a MongoDB collection.

The extracted information is stored in an internal MongoDB database, that we will call the *analytics database* in the remaining of this document. The structure of this database is shown at Figure 3. The *QLQuery* collection is at the core of the analytics database. It corresponds to the information stored for each TyphonQL DML query. Its attributes are the following:

- *query*: the TyphonQL query string, as sent to the TyphonQL engine and captured by the monitoring mechanism of WP5.
- *type*: the type of query (select, insert, delete, update).
- *allEntities*: the list of *all* TyphonML entities involved in the query
- *mainEntities*: the list of the *main* TyphonML entities involved in the query. For instance, in the case of a *select* query, the main entities are those occurring in the *from* clause.
- *executionDate*: the date of execution of the query.
- *executionTime*: the execution time (i.e., duration) of the query.
- *modelVersion*: the TyphonML schema version at the time of executing the query.

[1] the time interval being configurable.

- *joins*: the list of conceptual join conditions occurring in the query, e.g., *a.products == p*.
- *selectors*: the list of attribute-based selection criteria, e.g., *product.name == 'myproduct'*.

The *QLNormalizedQuery* collection corresponds to the query categories. For each query category, the analytics database stores:

- the normalized query, i.e., the query where all input values (constants) have been replaced with placeholders, and the unnecessary whitespaces have been removed;
- the number of queries of this category that have been executed in the considered period.

In our running example, we started from 1007 generated query events (*QLQuery* instances), corresponding to 314 different query categories (*QLNormalizedQuery* isntances).

The *TyphonModel* collection relates to the successive versions of the TyphonML schema during the considered period. The *TyphonEntity* and *TyphonEntityHistory* collections include information related to the polystore entities, their size, their data manipulation usage and their evolution history.

The analytics database populated during Step 2 constitutes the main input of the next three steps, which respectively aim at:

- providing users with interactive visual analytics of the polystore data usage (Step 3);
- providing users with polystore reconfiguration recommendations for those query categories suffering from poor performance (Step 4);
- applying the reconfiguration recommendations selected by the user (Step 5).

These three next steps are presented and illustrated below.

2.3 Step 3: Visual analytics of polystore data usage

The main page of the visual analytics tool is depicted in Figure 4. This page provides the user with a general overview (1) of the polystore configuration and (2) of the polystore data usage at a coarse-grained level.

Polystore schema view

On the right-hand side of the main page, as depicted in Figure 5, the user can see the current schema configuration of the polystore. In our example, one can see that the polystore currently consists of two databases: a relational database including 5 entities (tables), and a document database consisting of 2 entities (collections). The size of a circle relates to the number of records (rows, documents) in the corresponding database/entity.

By clicking on the relational database, the user can zoom and get more details about the entities it includes, as shown in Figure 6. Here, we see that the database includes two large entities (in this case, tables): *User* and *Address*; as well as three smaller tables: *CreditCard*, *Product* and *OrderProduct*.

By clicking on the document database, the user can see (Figure 7), that it includes two collections of a similar size: *Comment* and *Review*.

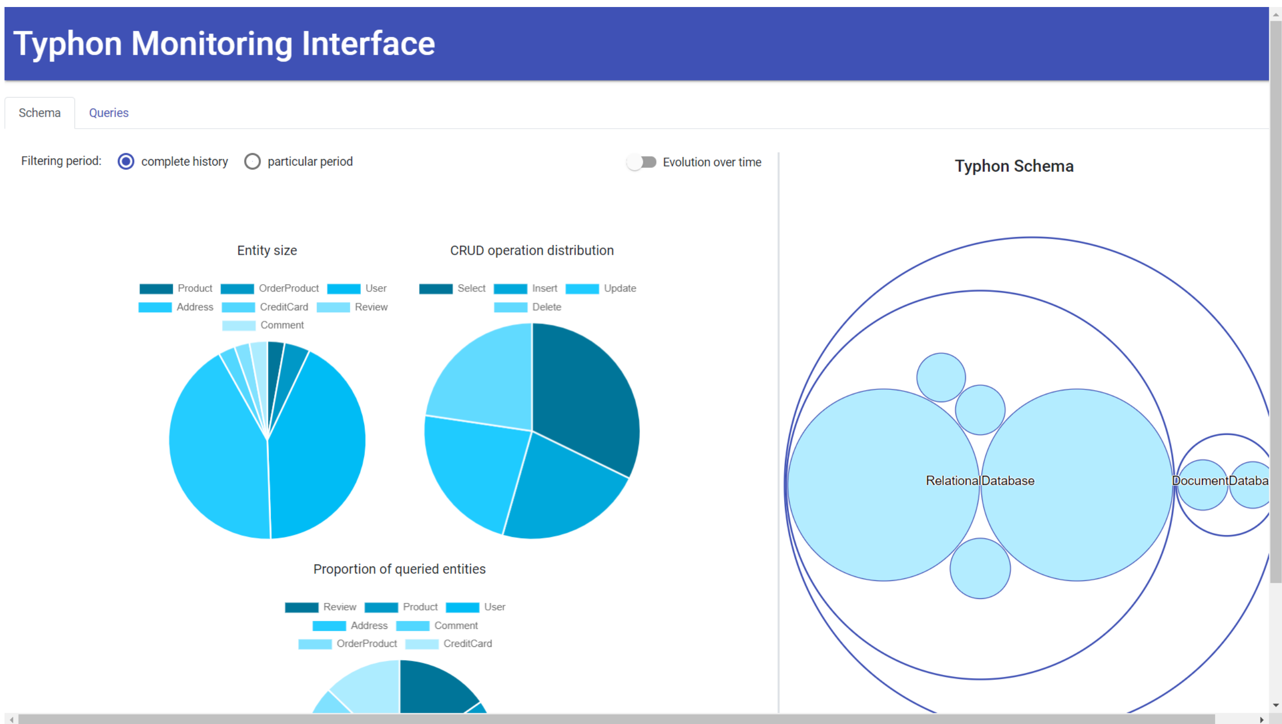

Figure 4: Main view of the continuous evolution tool

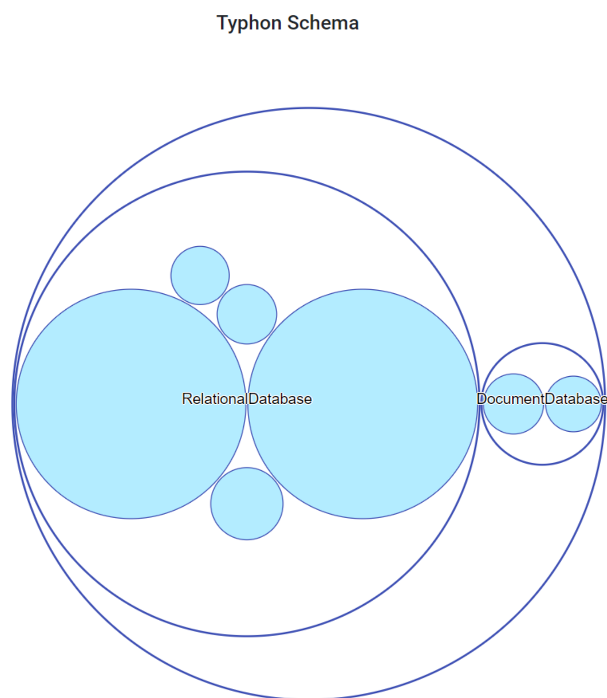

Figure 5: Global schema view of the polystore

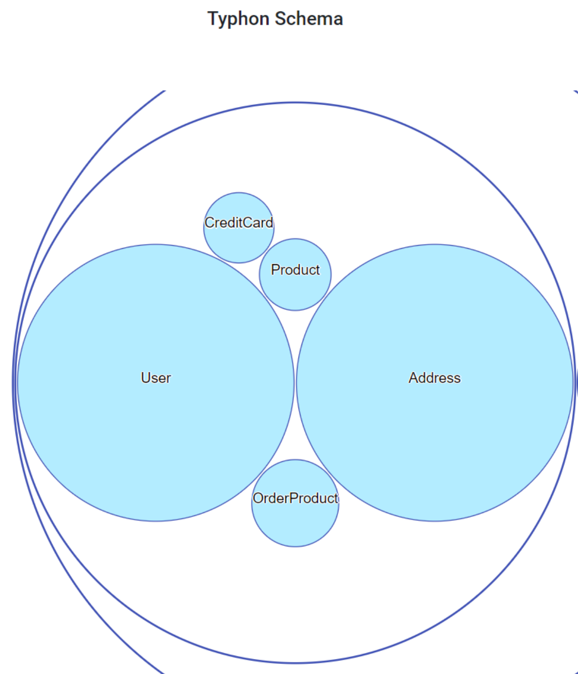

Figure 6: Schema view of the relational database

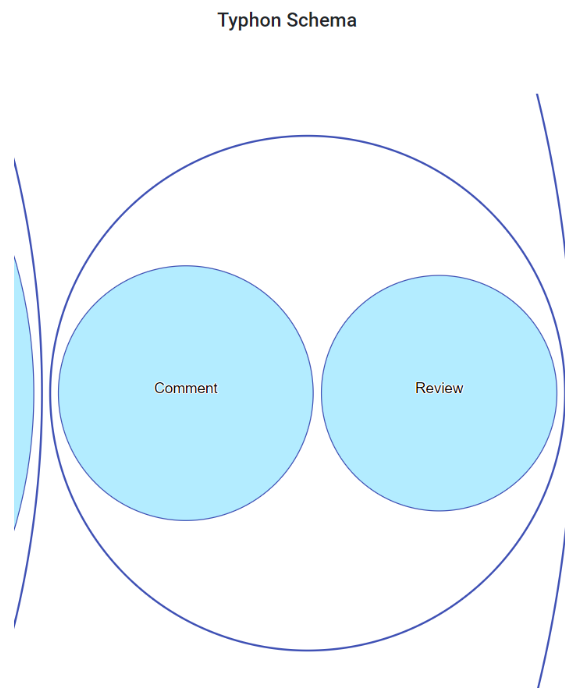

Figure 7: Schema view of the document database

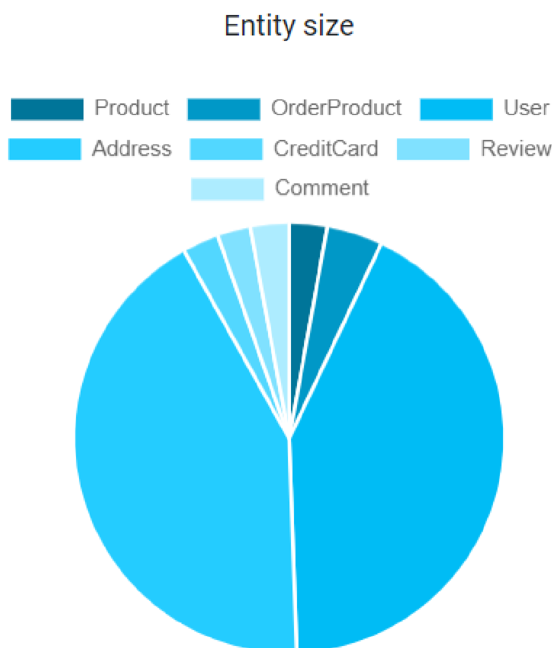

Figure 8: Entity size view

Polystore entities view

The tool provides the user with a global overview of the current size of the polystore entities, as shown in Figure 8. One can see that entities *User* and *Address* are the two largest entities of the polystore, in terms of number of records. Positioning the mouse pointer on a given entity would provide the user with the exact number of records for this entity.

The evolution of the entity size over time is also provided. The user can select the entities of interest, and the tool then shows the evolution of the size of the selected entities over time. For instance, at Figure 9 reveals that the size of entity *User* has rapidly increased. It has indeed recently jumped from 1.000 to 10.000 records.

Another visual metric concerns the volume and distribution CRUD operations that have been applied to the polystore during the considered period. For instance, Figure 10 shows a pie chart reflecting the distribution of CRUD operations executed (select, insert, delete, update). One can see that the CRUD operations are more or less equally distributed, as it can be expected from our random query event generation. Note that positioning the mouse pointer on a given CRUD operation allows to see the exact number of query occurrences of this type.

Polystore CRUD operations view

A similar metric is provided for the distribution CRUD operations by TyphonML entity. For instance, Figure 11 shows that CRUD operations are more or less equally distributed among the 7 polytore entities. Again, positioning the mouse pointer on a given entity allows to see the exact number of queries involving this entity.

The distribution of executed queries (by CRUD operation or by entity) can also be shown for a particular period of time, chosen by the user, as depicted in Figure 12.

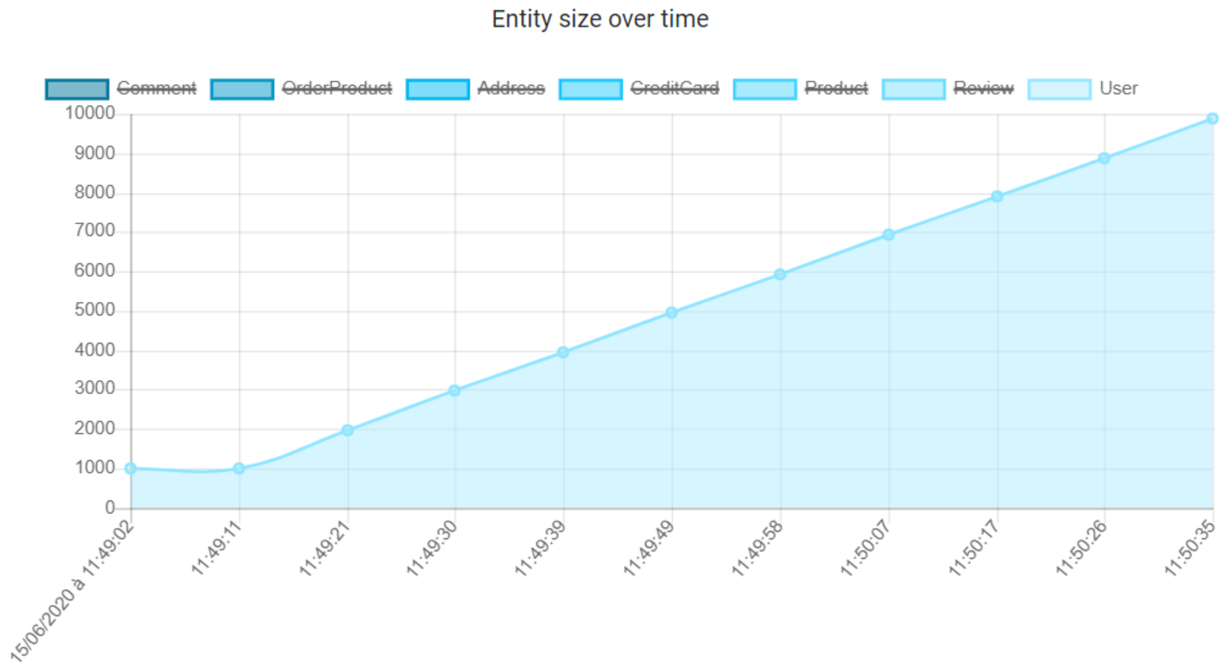

Figure 9: Size of the (selected) polystore entities over time, expressed in number of records

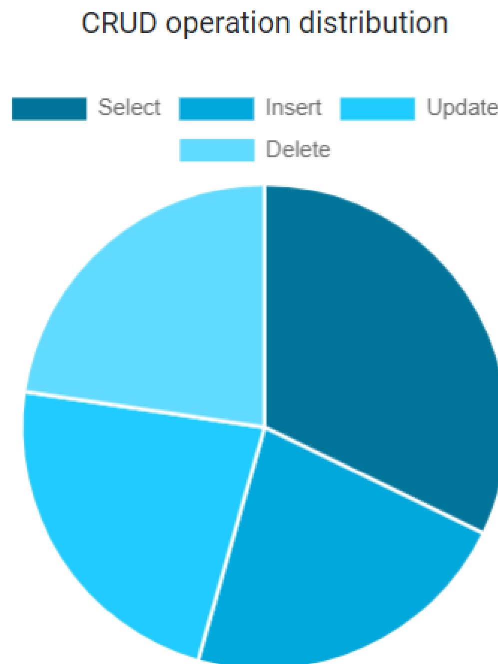

Figure 10: Distribution of CRUD operations applied to the polystore

The user can also look at the evolution of the number of CRUD operations executed over time, at the level of the entire polystore. Figure 13 shows an example of such an evolution, where we can see a peak at 120 select queries at a certain point.

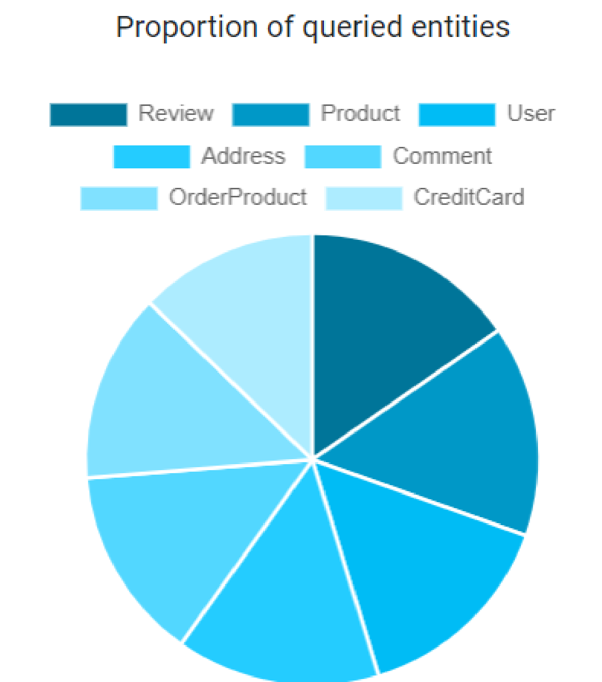

Figure 11: Distribution of queries among the polystore entities

A similar visual analytics is proposed at the level of polystore entities. One can see, for each entity, the evolution of the number of queries manipulating it over time. Figure 14 gives an example of such a visual report, where the user can either see the trend for all entities, or select the entity/entities of interest.

Polystore queries view

The user can then have a finer-grained look at the TyphonQL queries executed by the TyphonQL engine on the polystore. In the query view, the tool provides the user with two searchable lists:

- the list of the most frequent query categories, in decreasing order of number of occurrences, as shown on the left of Figure 15;
- the list of slowest queries, in decreasing order of execution time, as shown on the right of Figure 15.

Note that the user can also get the same lists, by considering a particular period of time. She can also search for particular queries using the search bar.

By looking at the query view, the user can figure out that the most frequent query category corresponds to the following TyphonQL query, that deletes a user based on his id:

```
delete User x0 where x0.id == "?"
```

A total of 39 occurrences of this query category where executed, with an average execution time of 405 milliseconds.

The following TyphonQL query was the slowest query during the entire considered period:

```
from Address x0, User x1 select x0, x1 where x0.user == x1 and x0.country == "w0Jn9A"
```

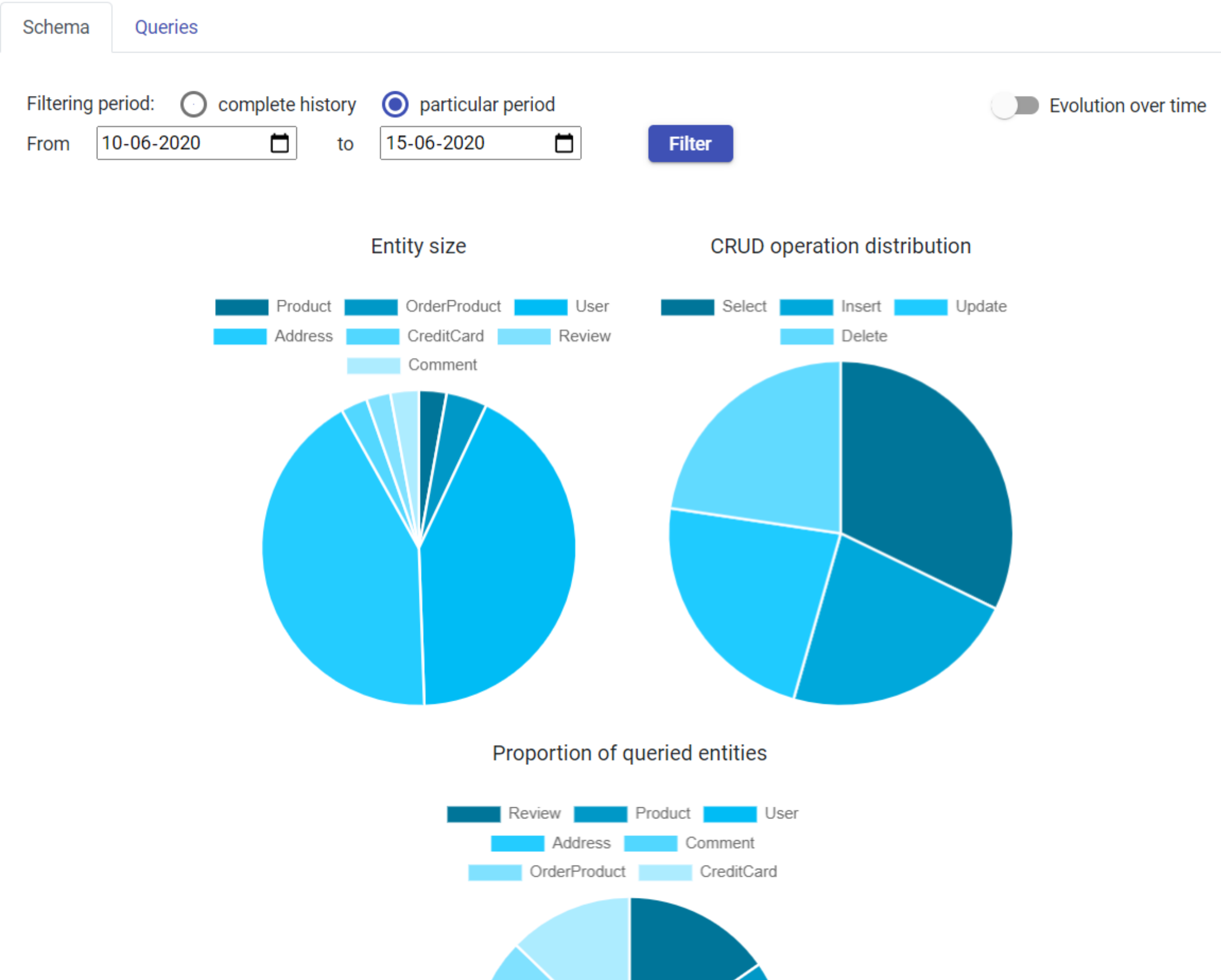

Figure 12: Distribution of CRUD operations and entity accesses during a particular period of time

Its execution took 8.670 milliseconds (fictitious duration).

By clicking on the *DETAILS* button, the user can get finer-grained information about the category of a particular query. For instance, at Figure 16, one can observe that the duration of the slowest query shown above used to be shorter in the past, but that it increased recently. On the right of the page, the tool provides links to the entities involved in the inspected query. In this case, the query involves a join between entities *Address* and *User*.

By inspecting entity *Address*, shown in Figure 17, one can observe that its size has dramatically increased in the considered period of time. At the beginning of the period, it included 1.000 records, while it currently includes 10.000 records. Entity *User* followed exactly the same trend in terms of size, as shown at Figure 18.

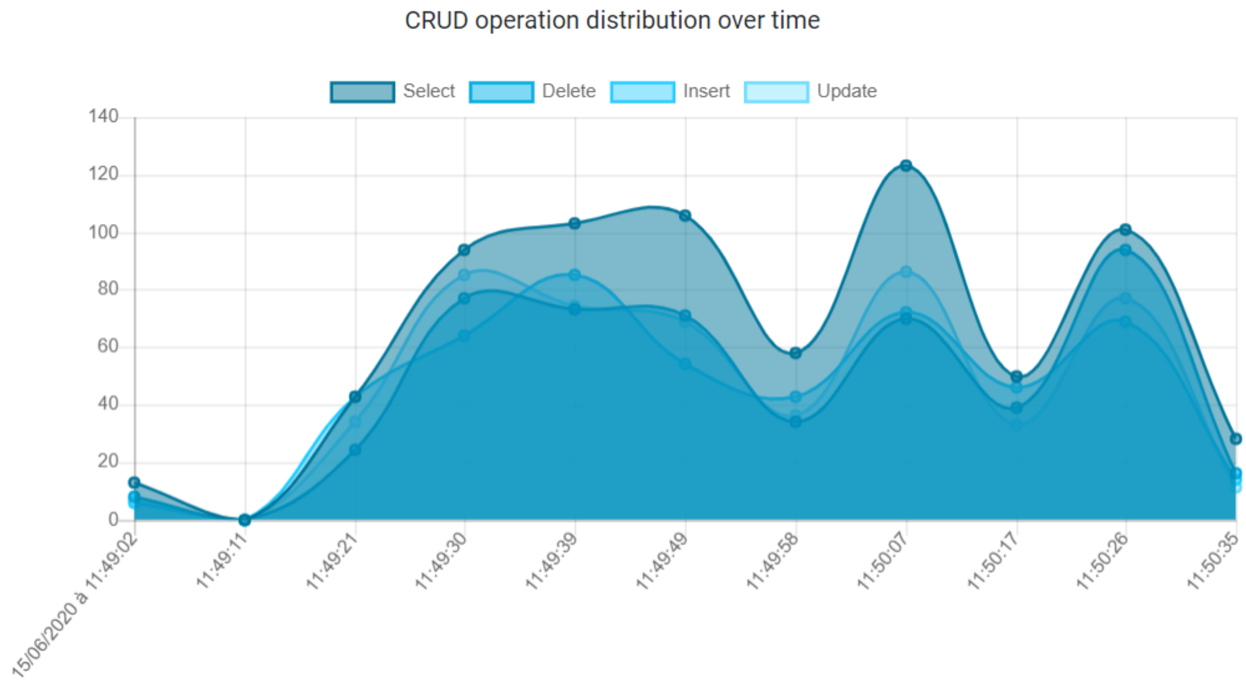

Figure 13: Distribution of CRUD operations over time, expressed in number of queries

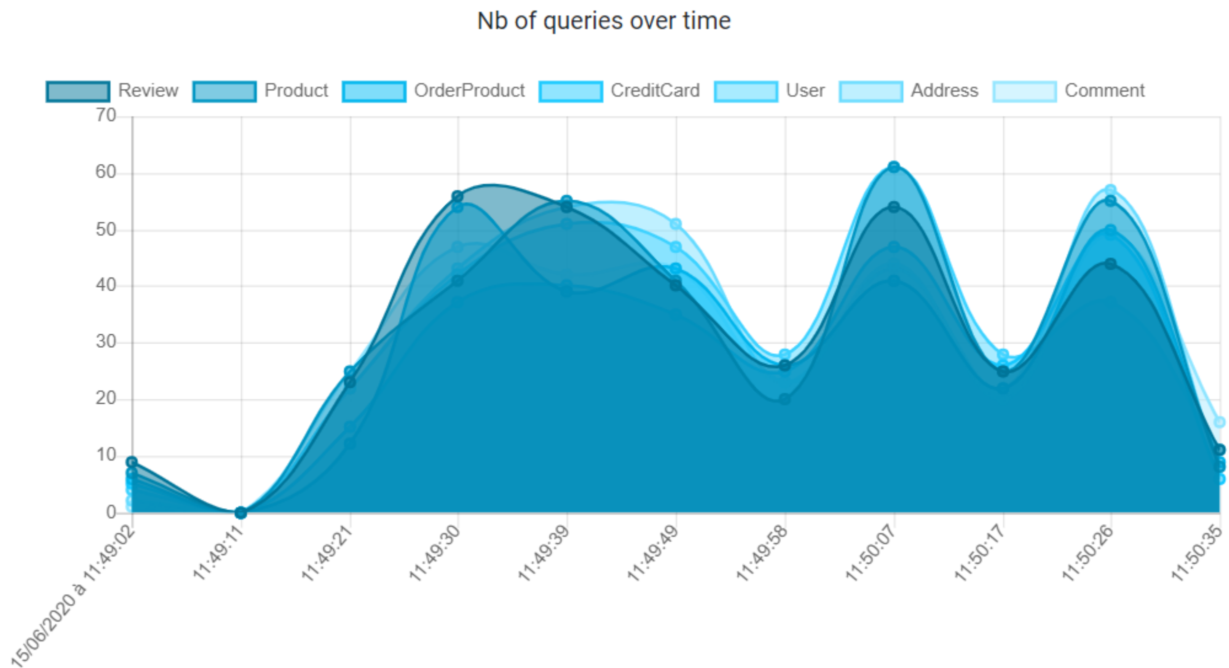

Figure 14: Number of queries accessing the (selected) polystore entities over time

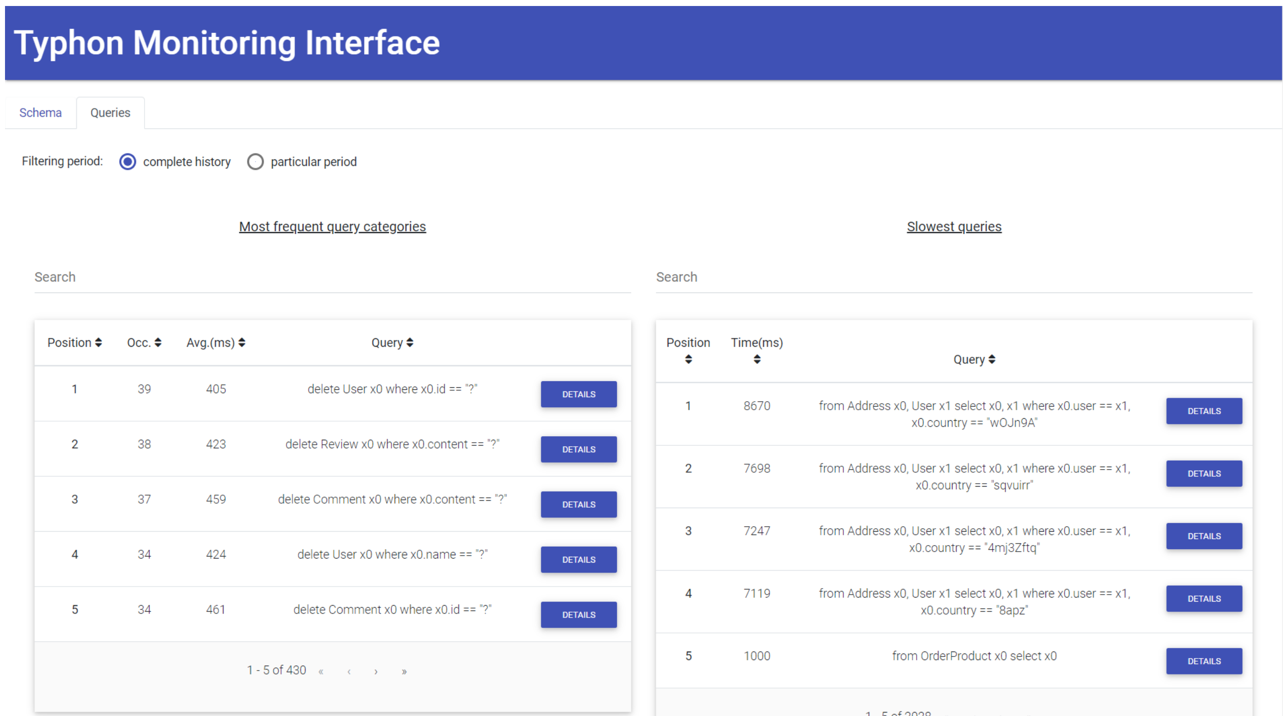

Figure 15: General overview of the most frequent query types (left), and of the slowest queries (right)

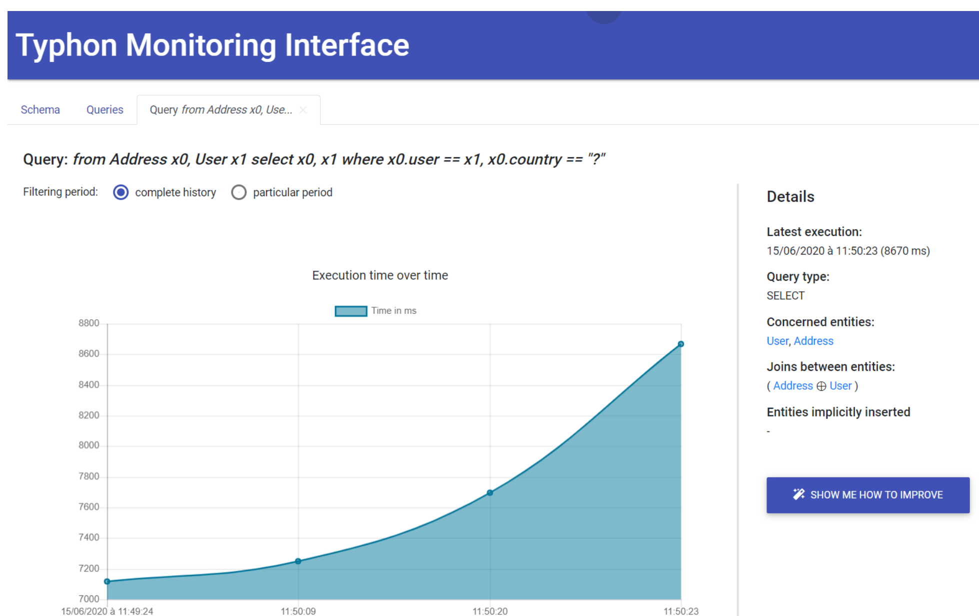

Figure 16: Duration of a particular query type over time

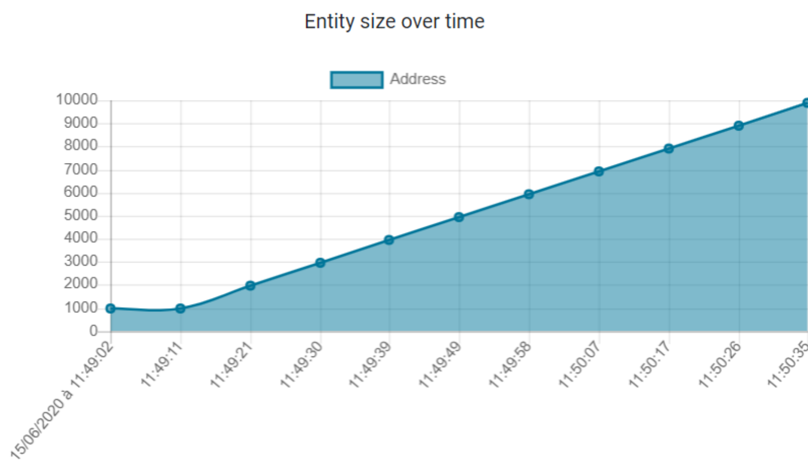

Figure 17: Evolution of the size of entity Address over time

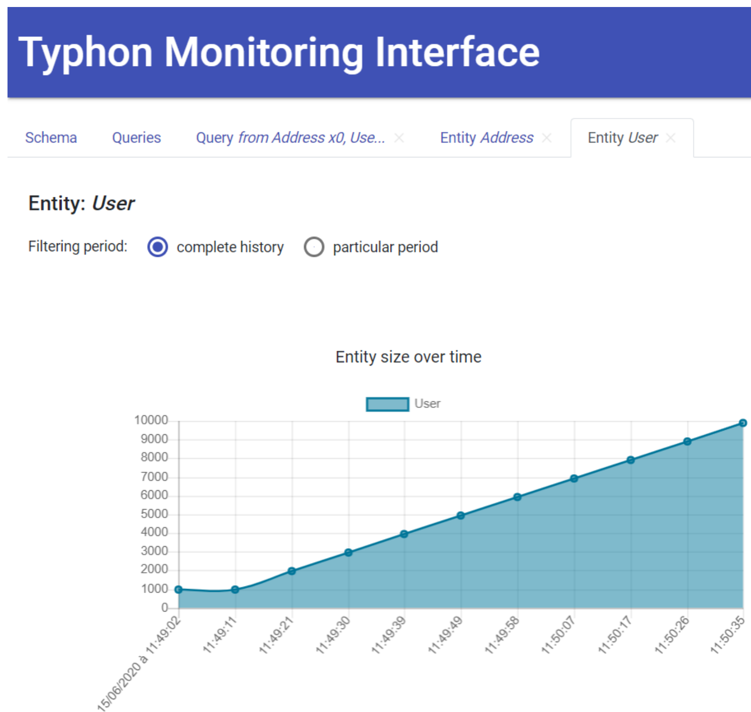

Figure 18: Evolution of the size of entity User over time

2.4 Step 4: Recommending polystore schema reconfigurations

When inspecting a particular (slow) query, the user can ask the tool for recommendations on how to improve the execution time of the query. When possible, the tool then recommends polystore schema reconfigurations, in the form of a menu with clickable options, including one of several recommendations. Some of the provided recommendations may be mutually-exclusive, which means that they cannot be selected together in the menu.

The example recommendations shown in Figure 19 correspond to the slow TyphonQL query presented above. This slow query involves a join between entities *User* and *Address* and a selection operator based on the value of the *Address.country* attribute. In this case, the continuous evolution tool recommends two possible, non-exclusive schema reconfigurations that respectively consist in:

- defining an index on column *AddressDB.country*, which maps with attribute *Address.country*;
- merging entity *Address* into entity *User*, via the one-to-one relation "Address.user" that holds between them.

By positioning the mouse pointer on the *information* icon, the user can get further information about the expected positive impact of the recommended schema change on the execution time of the query.

Adding an index on a column c is a well known technique to speed up a query including an equality condition on c in its where clause. In the particular case of the considered query, attribute *Address.country* is used in an equality condition. As we can observe in the TyphonML model of Figure 2, there is no index defined on table *AddressDB*, mapped with entity *Address*. Hence the recommendation made to the user to define an index on column *AddressDB.country*.

Merging two entities into a single entity constitutes another recommendation that allows to avoid a costly join condition in a slow query. In our example, the recommendation to merge entity *Address* into entity *User* is

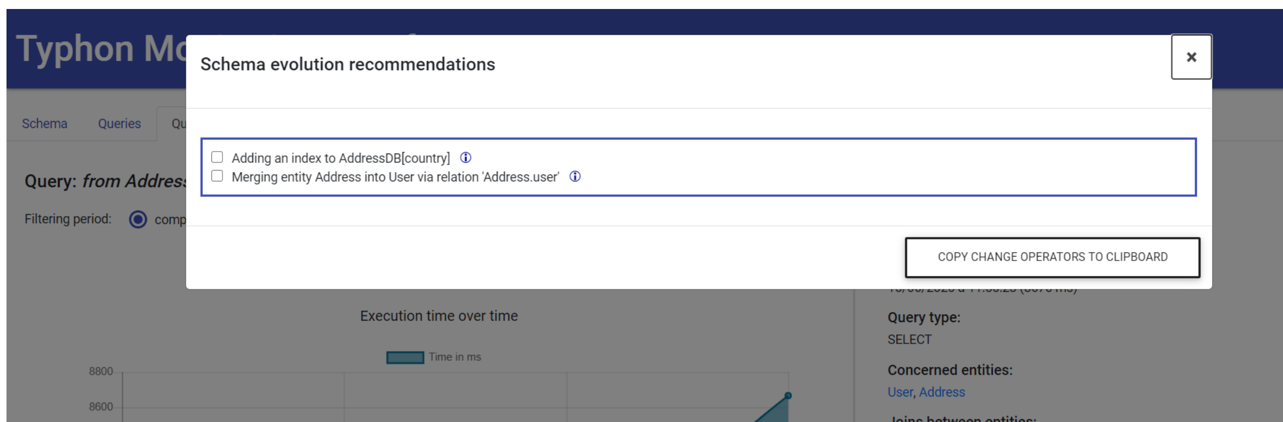

Figure 19: Schema evolution recommendations related to a particular (slow) query

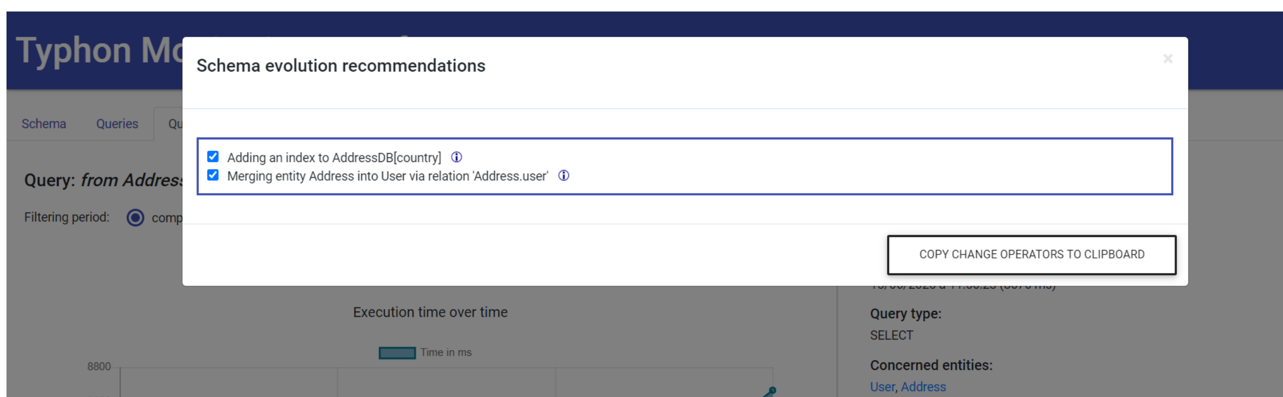

Figure 20: Selection of the schema evolution recommendations to apply to the polystore

motivated by the fact that the two entities are linked together via a one-to-one relationship (thus have the same number of records), and that both entities rapidly grow in terms of size, making the join condition slower and slower.

Another possible recommendation (not relevant in our example) is to migrate an entity from a database to another. For instance, let us assume that entity *Address* would be mapped to a document database (a MongoDB collection), while entity *User* would be mapped to a relational table. The join condition of the considered query would probably be even slower, since it would involve to separately query two different databases, and then to aggregate the results in the form of a join. In this case, our tool would provide as possible recommendation to migrate one of the entities (e.g., *Address*) to the other database. But this would be an exclusive choice with the merge recommendation, since the latter would include the migration of the entity.

2.5 Step 5: Applying the selected recommendations

Using the option menu, the user may then choose which evolution recommendation(s) (s)he wants to actually follow by selecting the desired option(s), as shown in Figure 20. Once this selection has been done by the user, the user can click on the *copy change operators to clipboard* button. The tool will then automatically generate the list of schema evolution operators corresponding to the selected recommendations (see Figure 21). Those

```
changeOperators  [
     AddIndex {table 'AddressDB' attributes ('Address.country') }
    merge entities User Address as 'Address.user'
 ]
```

Figure 21: Schema evolution operators corresponding to the recommendations selected at Figure 20

operators are expressed according to the TyphonML textual syntax (TML). So the user can simply paste the operators from the clipboard to the TML file of his TML schema, and then invoke the schema evolution tool via the Typhon API by passing the modified TML file as input.

The schema evolution tool will then apply the recommended schema reconfigurations to the polystore. This includes the adaptation of the polystore TyphonML schema, of the underlying native structures and of the data instances. Upon request of the user, the query evolution tool (presented in deliverable D6.4 [7]) can support the adaptation of existing TyphonQL queries

The schema evolution tool will produce the new TyphonML schema, where attribute *country* has now been indexed, and where entity *Address* has been merged into entity *User*. The resulting TyphonML model as shown in Figure 22 (in TML format).

Thanks to the query evolution tool, the slowest TyphonQL query category identified at Step 3:

```
from Address x0, User x1 select x0, x1 where x0.user == x1 and x0.country == "?"
```

can be automatically adapted, resulting in the following output TyphonQL query:

```
from User x1 select x1 where x1.country == "?"
```

The output query does not include any join condition on two entities. Its where clause includes an equality condition on an attribute that is now indexed. The performance of the query should therefore be significantly better than in the initial situation.

```
entity Review{
    id : string[32]
    content : string[32]
    product -> Product[1]
}

entity Product{
    id : string[32]
    name : string[32]
    description : string[32]
    reviews :-> Review."Review.product"[0..*]
}

entity OrderProduct{
    id : string[32]
    product_date : string[32]
    totalAmount : int
    paidWith -> CreditCard[1]
}

entity User{
    id : string[32]
    name : string[32]
    streetName: string[32]
    streetNumber: string[32]
    zip: string[32]
    city: string[32]
    country: string[32]
    comments :-> Comment[0..*]
    paymentsDetails :-> CreditCard[0..*]
    orders -> OrderProduct[0..*]
}

entity Comment{
    id : string[32]
    content : string[32]
    responses :-> Comment[0..*]
}

entity CreditCard{
    id : string[32]
    number : string[32]
    expiryDate : string[32]
}
```

```
relationaldb RelationalDatabase{
    tables{
        table {
            UserDB : User
            index  userIndex{
                attributes ('User.country')
            }
            idSpec ('User.name')
        }

        table {
            ProductDB : Product
            index productIndex{
                attributes ('Product.name')
            }
            idSpec ('Product.name')
        }

        table {
            CreditCardDB : CreditCard
            index creditCardIndex{
                attributes ("CreditCard.number")
            }
            idSpec ("CreditCard.number")
        }

        table {
            OrderDB : OrderProduct
            index orderIndex {
                attributes ("OrderProduct.id")
            }
            idSpec ("OrderProduct.id")
        }
    }
}

documentdb DocumentDatabase{
    collections{
            CommentsDB : Comment
            ReviewDB : Review
    }
}
```

Figure 22: TyphonML model obtained after applying the selected recommendations

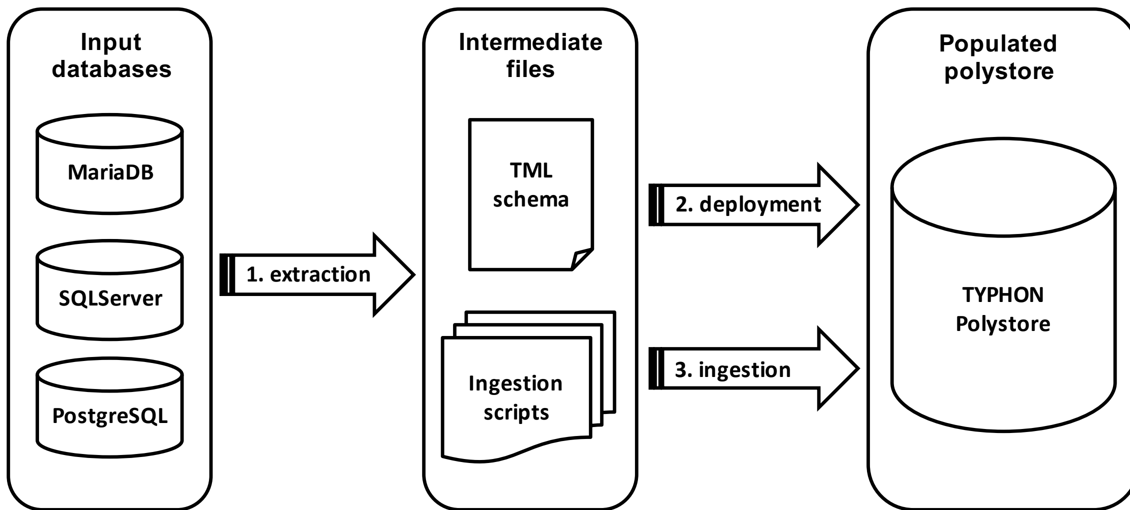

Figure 23: Overview of the process followed by the data ingestion tool

3 Data Ingestion Tool

The Data Ingestion tool aims to ease the adoption of the Typhon innovative technologies. It allows one to ingest data from (a set of) pre-existing relational database(s) into a Typhon polystore.

The data ingestion process relies on fours steps, as depicted in Figure 23:

- **Step 1 - extraction:** The tool first reverse-engineers the relational database schema of each input database, in order to produce a TyphonML schema. It also generated a set of data ingestion scripts allowing to transfer the data from the input relational database(s) towards the polystore, as soon as the latter will be deployed.
- **Step 2 - deployment:** The user takes the automatically extracted TML schema, and uses as starting point to manually deploy a new (empty) Typhon polystore. This deployment step can be done by means of the tools provided by Work Package 3.
- **Step 3 - ingestion:** The user can then execute the generated data ingestion scripts in order to populate the freshly created polystore with the data extracted from the input relational databases.

3.1 Step 1: Extraction

The extraction phase mainly consists in extracting the data structures (schemas) of the relational databases given as input, and to abstract those data structures into a TyphonML schema. This schema abstraction process is achieved according to the following abstraction rules.

- each table including at least one non-foreign key column becomes a conceptual entity;
- each non-foreign key column of a table becomes an attribute of the corresponding entity;
- each foreign-key becomes a one-to-many relationship;
- each table that only consists in two foreign keys referencing respectively table t_1 and table t_2, becomes a many-to-many relationship between the corresponding entities;
- all relational schema elements including identifiers and indexes are also translated into corresponding TyphonML schema constructs.

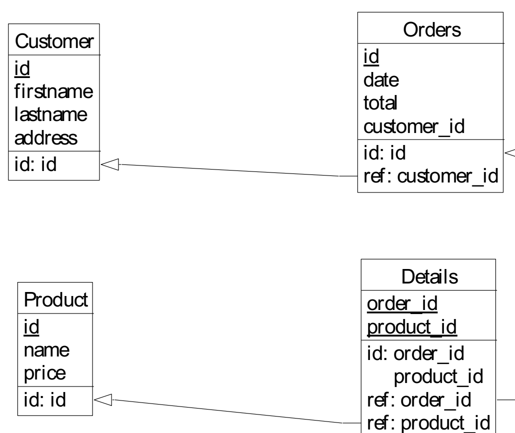

Figure 24: Example schema of an input relational database

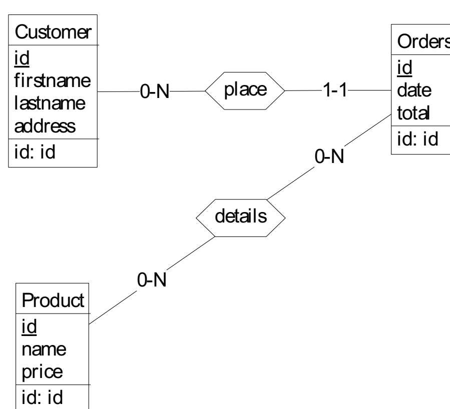

Figure 25: Conceptual abstraction of input schema of Figure 24

As an example, let us consider the input relational schema of Figure 24. This schema includes 4 tables: *Customer*, *Orders*, *Product* and *Details*.

The input schema of Figure 24 would be abstracted as three entities, as shown in the conceptual schema of Figure 25. Tables *Customer*, *Orders* and *Product* have been translated into corresponding entities. Table *Details* has been abstracted as a many-to-many relationship. The foreign keys in table *Orders* referencing table *Customer* has been abstracted as a one-to-many relationship between the corresponding entities.

This conceptual abstraction will lead to the production of the TyphonML schema given in Figure 26. This schema can be used as starting point of the deployment step (Step 2).

In order to connect to the input databases, the data ingestion tool requires the user to specify the required URL and credentials. This information must be contained in a configuration file ("extract.properties"). In this file, one can specify the connection information of one or several relational databases.

The following extraction parameters can be specified, for each input relational database:

- URL : the JDBC URL necessary to connect to the database.
- DRIVER : the JDBC driver necessary to connect to the database.
- USER : a user login with reading permissions.

```
entity Orders{
    "id": string[32]
    "date" : date
    "total" : float
    "Customer" -> Customer[1]
    "Product" -> Product[0..*]
}

entity Customer{
    "id" : string[32]
    "firstname" : string[32]
    "lastname" : string[32]
    "address" : string[32]
}

entity Product{
        "id": string[32]
        "name": string[32]
        "price": float
        "Orders" -> Orders[0..*]
}

relationaldb RelationalDatabase{
    tables{

        table {
            "Orders" : "Orders"
            idSpec ("Orders.id")
        }

        table {
            "Customer" : "Customer"
            idSpec ("Customer.id")
        }

        table {
                "Product" : "Product"
                idSpec ("Product.id")
        }
    }
}
```

Figure 26: TyphonML model extracted from the input relational schema of Figure 24

```
1  URL=jdbc:mysql://localhost:3306/test
2  DRIVER=org.mariadb.jdbc.Driver
3  USER=root
4  PASSWORD=example
5  SCHEMA=test
6  DOCUMENT_SPLIT=Employee.history,Department.nbOfEmployees
7
8  #URL2=jdbc:sqlserver://localhost;databaseName=ACCS_DB;integratedSecurity=true;
9  #DRIVER2=com.microsoft.sqlserver.jdbc.SQLServerDriver
10 #USER2=
11 #PASSWORD2=
12 #CATALOG2=ACCS_DB
13 #SCHEMA2=dbo
14 #DOCUMENT_SPLIT2=test.xml_
15
16 #URL3=jdbc:mysql://localhost:3306/RelationalDatabase
17 #DRIVER3=org.mariadb.jdbc.Driver
18 #USER3=root
19 #PASSWORD3=example
20 #SCHEMA3=RelationalDatabase
21
22
23
24 MAX_SQL_QUERIES_PER_FILE=100000
25 MAX_JSON_RECORDS_PER_FILE=10000
26
```

Figure 27: Example of configuration for the extraction step

- PASSWORD : the user password.
- SCHEMA : the name of the input database schema name to connect.
- DOCUMENT_SPLIT : this parameter is optional. It allows the user to specify the database column(s) that will be migrated in a document database of the polystore (currently MongoDB). All other columns will be migrated into a relational database of the polystore (default behavior).
- MAX_SQL_QUERIES_PER_FILE: this parameter specifies the maximal number of insert queries per data ingestion script. This parameter can be useful in presence of a huge amount of records to ingest to the polystore.
- MAX_JSON_RECORDS_PER_FILE: in case the DOCUMENT_SPLIT parameter is used, the plugin will generate JSON files containing the JSON records to ingest in the polystore document database. Like the relational data ingestion scripts, the user can specify the maximal number of json records per ingestion file.

In the case of several input relational databases, the user can use a suffix for each of the above parameters. Following the pattern *PARAMETER#DB*, i.e *URL2* will be the URL of the second database, *URL3* will be the URL of the third database,. . .

Figure 27 provides an example of configuration file. In this example, at line 6, the user indicates that column *history* of table *Employee* must be migrated in a document database in the polystore, as well as column *nbOfEmployees* of table *Department*.

```
1 RELATIONAL_DB_URL=jdbc:mysql://localhost:3306/RelationalDatabase
2 RELATIONAL_DB_DRIVER=org.mariadb.jdbc.Driver
3 RELATIONAL_DB_USERNAME=root
4 RELATIONAL_DB_PWD=example
5
6 DOCUMENT_DB_URL_WITH_AUTH=mongodb://admin:admin@localhost:27018
7 DOCUMENT_DB_NAME=DocumentDatabase
```

Figure 28: Data ingestion parameters

3.2 Step 2: Deployment

The polystore deployment step simply consists in deploying a new polystore by using the TyphonDL tools (WP3). This process takes as input the TyphonML schema automatically extracted at Step 1. We refer to the WP3 deliverables for more details about the deployment process and supporting tools.

3.3 Step 3: Ingestion

Once the new target polystore has been created and deployed, the last step consists in executing the data ingestion scripts generated at Step 1.

The execution of those scripts also requires to specify the URL and credentials required to connect the relational database of the polystore and, optionnaly, to the document database of the polystore. This information must be defined in the *inject.properties* file. Figure 28 gives an example of structure for this configuration file. The following parameters are required:

- RELATIONAL_DB_URL: the JDBC URL required to connect to the relational database of the target polystore.
- RELATIONAL_DB_DRIVER: the JDBC driver required to connect to the relational database of the target polystore[2]
- RELATIONAL_DB_USERNAME: a user login with reading permissions.
- RELATIONAL_DB_PWD: the user password.
- DOCUMENT_DB_URL_WITH_AUTH: the URL to connect to the document database of the target polystore[3].

Once the configuration file has been edited, the data ingestion process can be launched, depending on the operating system:

- For Windows: *sql_extract.bat -inject inject.properties output/data*
- For Linux: *sql_extract.sh -inject inject.properties output/data*

Where *inject.properties* is the configuration file described above and *output/data* is the directory containing the SQL/JSON migration scripts generated during the extraction phase (Step 1). Once the data ingestion is completed, the polystore is populated and ready to use.

[2] At the time of writing, only MariaDB is supported as target relational database.
[3] At the time of writing, only MongoDB is supported as target document database.

4 Summary of WP6 contributions

In this section, we summarize the contributions of Work Package 6, by assessing the coverage of the technical requirements (see Table 1) and of the use case requirements (see Table 2) specified in deliverable D1.1.

Regarding the technical requirements:

- **Requirement 57** is covered. A dedicated schema change API, callable from the TyphonML editor or from the user has been implemented. This schema evolution and data migration component was presented in deliverable D6.3 [6].
- **Requirement 58** is covered. A query evolution tool has been developed. It allows users to adapt TyphonQL queries to an evolving TyphonML schema, when possible, i.e., in the case of data-preserving schema changes (e.g., rename attribute, merge entities, etc.). The query evolution tool was presented in deliverable D6.4 [7].
- **Requirement 59** is partially covered. The query evolution, presented in deliverable D6.4 [7], is indeed able to identify TyphonQL queries that have become invalid due to non-data-preserving schema changes (e.g., delete entity) applied to the polystore schema. Those invalid queries are marked as *broken* in the output set of queries. In some cases (e.g., change attribute type), the tool may simply mark an output query with a *warning*, which means that the context of the TyphonQL query requires manual inspection.
- **Requirement 60** is covered. Data-preserving, intra-paradigm schema changes are propagated at the level of the data instances, by exploiting the power of TyphonQL for (1) reading data according to the source polystore schema, (2) adapting the native data structures, and (3) (re)writing data according to the target polystore schema. The schema evolution tool was presented in deliverable D6.3 [6].
- **Requirement 61** is covered. Cross-paradigm schema change scenarios are supported through the *migrate entity* schema evolution operator, presented in deliverable D6.3 [6]. This operator allows the automatic migration of a TyphonML entity from a database platform to another (e.g., from a relational to document database, or vice versa). Here again, the schema evolution tool benefits from the power of TyphonQL, without directly accessing the native databases involved in such migration scenarios.
- **Requirement 62** is covered. The continuous evolution tool presented in the present deliverable (D6.5) suggests data-preserving polystore reconfigurations, based on database usage performance monitoring.

Regarding the use case requirements:

- **Requirement 77** is covered; it corresponds to technical requirement 60.
- **Requirements 78** and **79** are covered; they correspond to technical requirement 61.
- **Requirement 80** is partially covered in the sense that the migrate operator currently considers the TyphonML entity as granularity level. In other words, it migrates the data of one entity at a time.
- **Requirement 81** is partially covered in the sense that the migrate operator makes use of TyphonQL to read the input data, to create the target native data structures and to write data to those target structures. Therefore, as soon as TyphonQL will support a future new data source, the migrate operator should still be able to migrate data from this data source, possibly with minor modifications.
- **Requirement 82** is covered; it relates to technical requirement 61.
- **Requirement 83** is not covered. It would require to exploit the pre-authorization mechanisms of Work Package 5 to temporarily prevent the execution of those TyphonQL queries that would impact/be impacted by the real-time data migration process.
- **Requirements 84-85** are not covered. They could, however, be covered in the future, in case TyphonQL would support Hive as native database backend.

Number	Requirement	Priority	Status
57	The schema evolution methodology and tools shall support the propagation / impact analysis of TyphonML polystore schema changes via a dedicated schema change API, callable from the TyphonML editor	SHALL	Covered
58	The Hybrid Polystore Query Evolution Tools shall support the automated propagation of data-preserving TyphonML polystore schema changes to related TyphonQL queries (consistency preservation)	SHALL	Covered
59	The Hybrid Polystore Query Evolution Tools may assess the impact of non-data-preserving polystore schema changes to the other polystore components, by identifying invalid TyphonQL queries (inconsistency detection)	MAY	Partially covered
60	The Hybrid Polystore Data Migration Tools shall support intra-paradigm data migration, i.e., in reaction to data-preserving modifications applied to the TyphonML polystore schema, without changing the database platform(s)	SHALL	Covered
61	The Hybrid Polystore Data Migration Tools shall support cross-paradigm data migration, i.e., in reaction to a data-preserving reconfiguration of the polystore involving a database platform change (e.g., migrating some polystore data from a relational DB to a NoSQL DB, or vice versa)	SHALL	Covered
62.	The Hybrid Polystore Continuous Evolution Tools shall suggest data-preserving reconfigurations of the polystore based on database usage performance monitoring	SHALL	Covered

Table 1: Coverage of technical requirements related to Work Package 6

Number	Requirement	Priority	Status
77	The polystore data migration tools shall allow to migrate data stored under different database technologies (SQL, NoSQL) within the polystore	SHALL	Covered
78	The polystore data migration tools shall allow to migrate the structure of the database instances stored under different database technologies (SQL, NoSQL) within the polystore	SHALL	Covered
79	The polystore data migration tools shall allow the bidirectional migration of data between different database technologies within the polystore	SHALL	Covered
80	The polystore data migration tools should allow the bidirectional migration of parts of the structure of the database instance between different database technologies within the polystore	SHOULD	Partially covered
81	The polystore data migration tools should support migrating data from future new data sources	SHOULD	Partially covered
82	The polystore data migration tools shall ensure data is not lost during data migration	SHALL	Covered
83	The polystore data migration tools may support data migration in real-time	MAY	Not covered
84	The polystore data migration tools may support migration from SQL database to Hive structures	MAY	Not covered
85	The polystore data migration tools may support migration from XML files to Hive structures	MAY	Not covered
86	The polystore data migration tools may support migration from text files to Hive structures	MAY	Not covered

Table 2: Coverage of use case requirements related to Work Package 6

Beyond the initial WP6 technical and use case requirements identified in deliverable D1.1, an additional WP6 tool has been developed upon request of our use case partners. This tool, presented in this deliverable, supports the automated ingestion of data from pre-existing relational databases into a newly deployed Typhon polystore.

5 Conclusions

In this deliverable, we have presented the Typhon continuous evolution tools that automatically suggest polystore reconfigurations to the user, by exploiting the monitoring mechanisms developed in Work Package 5. Starting from the capture and analysis of the polystore query events, the tool provides the user with visual analytics of the polystore data usage. The user can see, among others, which types of queries access which entities, which query categories are the most frequent, and which queries suffer from a lack of performance. The tool then makes, for each query category considered as problematic, possible schema reconfiguration recommendations aiming the speed-up the execution considered query. The user can then select the recommendations to follow, and then apply the associated schema evolution operators, using the components presented in our previous deliverables (see D6.3 [6] for schema change and data migration, and D6.4 [7] for query adaptation).

We have also presented an additional data ingestion tool, that we developed on request of our use case partners. This data ingestion tool supports the ingestion of data from pre-existing relational databases into a new polystore. As a first step, the data ingestion tool automatically extracts a TyphonML model M, abstracting the data structures of the input database(s). It also produces an executable data ingestion script that allows one to populate the polystore with the data contained in the input database. This ingestion script can be executed once the new polystore has been created and deployed using model M.

The main next steps in Work Package 6 include: (1) to continuously improve the WP6 tools according to the feedback we will receive from our use case partners, and (2) to further disseminate our research and development results to a wide audience, e.g., via the participation to international events and the submission of papers to international venues.

References

[1] Centrum Wiskunde & Informatica (CWI). D4.2 – Hybrid Polystore Query Language (TyphonQL), 2018.

[2] Centrum Wiskunde & Informatica (CWI). D4.3 – TyphonQL Compilers and Interpeters (Initial Version), 2018.

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

[4] The University of L'Aquila. D2.4 – TyphonML Modelling Tools, 2019.

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

[6] University of Namur. D6.3– Hybrid Polystore Data Migration Tools, 2019.

[7] University of Namur. D6.4– Hybrid Polystore Query Migration Tools, 2019.

[8] The University of York. D5.2 – data event organisation and representation report, 2018.

[9] The University of York. D5.3 – data event publishing and processing report, 2019.