Project Number 780251

D6.4 Hybrid Polystore Query Evolution Tools

Version 1.0
20 December 2019
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.

© 2019 Copyright in this document remains vested in the TYPHON Project Partners.

Project Partner Contact Information

<table>
<tr><td>Alpha Bank
Vasilis Kapordelis
40 Stadiou Street
102 52 Athens
Greece
Tel: +30 210 517 5974
E-mail: vasileios.kapordelis@alpha.gr</td><td>ATB
Sebastian Scholze
Wiener Strasse 1
28359 Bremen
Germany
Tel: +49 421 22092 0
E-mail: scholze@atb-bremen.de</td></tr>
<tr><td>Centrum Wiskunde & Informatica
Tijs van der Storm
Science Park 123
1098 XG Amsterdam
Netherlands
Tel: +31 20 592 9333
E-mail: storm@cwi.nl</td><td>CLMS
Antonis Mygiakis
Mavrommataion 39
104 34 Athens
Greece
Tel: +30 210 619 9058
E-mail: a.mygiakis@clmsuk.com</td></tr>
<tr><td>Edge Hill University
Yannis Korkontzelos
St Helens Road
Ormskirk L39 4QP
United Kingdom
Tel: +44 1695 654393
E-mail: yannis.korkontzelos@edgehill.ac.uk</td><td>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</td></tr>
<tr><td>OTE
Theodoros E. Mavroeidakos
99 Kifissias Avenue
151 24 Athens
Greece
Tel: +30 697 814 7618
E-mail: tmavroeid@ote.gr</td><td>SWAT.Engineering
Davy Landman
Science Park 123
1098 XG Amsterdam
Netherlands
Tel: +31 633754110
E-mail: davy.landman@swat.engineering</td></tr>
<tr><td>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</td><td>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</td></tr>
<tr><td>University of Namur
Anthony Cleve
Rue de Bruxelles 61
5000 Namur
Belgium
Tel: +32 8 172 4963
E-mail: anthony.cleve@unamur.be</td><td>University of York
Dimitris Kolovos
Deramore Lane
York YO10 5GH
United Kingdom
Tel: +44 1904 325167
E-mail: dimitris.kolovos@york.ac.uk</td></tr>
<tr><td>Volkswagen
Behrang Monajemi
Berliner Ring 2
38440 Wolfsburg
Germany
Tel: +49 5361 9-994313
E-mail: behrang.monajemi@volkswagen.de</td><td></td></tr>
</table>

Document Control

Version	Status	Date
0.1	Document outline	21 October 2019
0.4	First draft	30 November 2019
0.9	Full draft for partner review	16 December 2019
1.0	Updates from QA Review	20 December 2019

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 Context **3**

3 Query Migration **4**

3.1 Query Migration Process . . . 4

3.2 Query Migration Rules . . . 4

3.2.1 SMOs on Entity . . . 5

3.2.2 SMOs on Attribute . . . 8

3.2.3 SMOs on Relationship . . . 11

4 Example Query Migration Scenario **15**

5 Implementation **17**

5.1 Tool Description . . . 17

5.2 User Guide . . . 17

6 Conclusions **20**

Executive Summary

In the context of its Work Package 6, the TYPHON project aims to develop a methodology and technical infrastructure to support the graceful evolution of hybrid polystores, where multiple 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 third aspect of our evolution methodology, namely the automatically-supported adaptation of TyphonQL queries to an evolving TyphonML polystore schema. We present the general method and the tool we developed in order to support this query migration process.

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 deliverable D6.3, we have presented our *data migration* tool that aims at supporting the first two aspects, namely polystore schema evolution and related data migration. This data migration tool supports (i) the adaptation of the TyphonML model starting from a list of schema evolution operators, (ii) the adaptation of the underlying platform-specific data structures by exploiting the TyphonQL DDL operators, and (iii) the migration of the data from a source to a target schema, by exploiting the TyphonQL DML language.

In the present deliverable, we focus on the *query migration* step of our evolution methodology, the goal of which is to support the adaptation of TyphonQL queries to an evolving TyphonML polystore schema. We present the general method and the tool that we developed in order to support this query migration process.

1.1 Purpose of the deliverable

This document presents the work that has been done with respect to task 6.4 of Work Package 6, described as follows in the TYPHON Description of Work:

Task 6.4: This task aims to develop a method and automated support for adapting TyphonQL queries to an evolving polystore schema. The query adaptation mechanism may differ depending on the schema evolution scenario considered (as identified in Task 6.2). A generative approach will be used to propagate the evolution at the access API level. Transformational techniques will support the automated conversion of the TyphonQL queries.

1.2 Relationship to other TYPHON deliverables

The present deliverable presents a query migration tool that aims at adapting queries under polystore schema evolution.

Any polystore schema evolution scenario is expressed as a chain of Schema Modification Operators (SMOs). Those operators are fully specified in deliverable D6.2 [6], and are integrated in the TyphonML modeling language (presented in deliverables D2.3 [4] and D2.4 [5]).

The queries subject to migration are expressed in the TyphonQL language (presented in deliverables D4.2 [1] and D4.3 [2]).

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 this deliverable and of the query migration tool. A demo of this tool was given during the Typhon project meeting in Bremen (28-29 November 2019).

1.4 Structure of the deliverable

The remainder of this Deliverable is structured as follows:

- In Section 2, we briefly remind the context in which the *query migration* tool is used;
- Section 3 presents, for each TyphonML schema evolution operator, the query transformation rules followed by the *query migration* tool when adapting TyphonQL queries;
- In Section 4, we illustrate the use of the *query migration* tool, by considering a schema evolution scenario that involves a *chain* of schema evolution operators as well as a set of TyphonQL queries to migrate;
- We further discuss the implementation of our query migration tool, and we provide a brief user guide in Section 5;
- Section 6 provides concluding remarks and anticipates future work in Work Package 6.

2 Context

The query migration tool is used in the context of the evolution of a TYPHON polystore. This tool should be used *after* the schema evolution and data migration steps have been carried out. In other words, we make the assumption that:

- the current polystore schema has been evolved from a *source* schema to a *target* schema, using a chain of *schema evolution operators* (SMOs);
- the native data structures have been changed accordingly, using the TyphonQL DDL operators;
- the related data has been migrated towards the target polystore configuration, using the TyphonQL DML language.

When using the query migration tool, the user gives as input (1) the source TyphonML schema, (2) the set of schema modification operators applied to this source schema and (3) a set of existing TyphonQL queries, expressed on top of the source schema. This set of input queries actually includes two subsets of queries:

- queries that are not impacted by the schema evolution scenario, i.e., they remain valid with respect to the *target* chema;
- queries that are impacted by the schema evolution scenario, i.e;, they became invalid with respect to the *target* schema.

The query migration tool mainly focuses on the second category of queries. It aims to automatically transform those invalid queries, expressed on top of the *source* schema, into *equivalent* queries expressed on top of the *target* schema.

Depending on the schema evolution scenario considered, this query transformation is not always possible. For instance, if a TyphonML entity is deleted, any TyphonQL query that operates on this entity becomes invalid, and there is no way to transform it into an equivalent query expressed on top of the target schema.

The query migration returns as output a set of queries, with an annotating comment for each output query, with 4 possible cases:

- UNCHANGED: the input query has not been changed since it remains valid with respect to the target schema;
- MODIFIED: the input query has been transformed into an equivalent output query, expressed on top of the target schema;
- WARNING: the output query (be it unchanged or transformed) is valid with respect to the target schema, but it may return a different result set;
- BROKEN: the input query has become invalid, but it cannot be transformed into an equivalent query expressed on top of the target schema.

3 Query Migration

In this section, we first describe the general architecture of the *Query Migration Tool* (Section 3.1). Then, we present the different transformation rules that our Query Migration tool follows according to deliverable D6.2 [6] each schema modification operator (Section 3.2).

3.1 Query Migration Process

The process followed by the *Query Migration Tool* is shown at Figure 3.1. The tool takes three different inputs: (i) the source TyphonML polystore schema, (ii) a set of TyphonQL queries expressed on top of the source TyphonML polystore schema, and (iii) a list of Schema Modification Operators (SMOs) (see deliverable D6.2 [6]) applied to the source TyphonML polystore schema. Based on those inputs, our Query Migration tool aims at adapting the input TyphonQL queries to the target polystore schema, i.e., the schema obtained by applying the SMOs to the source polystore schema.

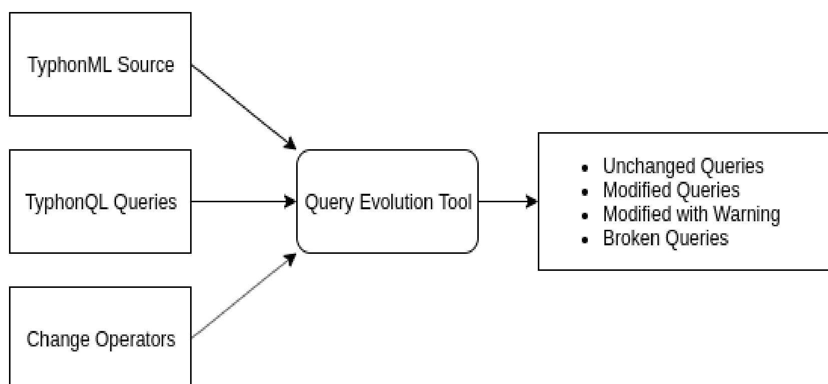

Figure 1: Overview of *Query Migration Process*

As output, a set of output TyphonQL queries is produced by the query migration tool. Each of the output queries belongs to one of the four possible categories of output queries: (i) Unchanged queries, (ii) Modified queries, (iii) Modified queries with Warning or (iv) Broken queries.

3.2 Query Migration Rules

In this section, we specify and illustrate the transformation rules, according to which a set of TyphonML-to-TyphonML schema transformations are propagated as corresponding Query Migration rules defined on top of the TyphonQL query language.

The Query Migration Tool is able to (i) identify queries that could not be transformed (*Broken* queries), (ii) identify queries that remain valid and do not require any adaptation (*Unchanged* queries), and (iii) adapt queries that can be transformed into equivalent queries expressed on top of the target schema (*Modified queries*). The tool also provides the user with useful warnings in case an output query has a slightly different behavior that the corresponding input query (*Modified with warning* queries).

For each schema modification operator (SMO), we successively present:

- the schema modification operator;

- the source and target polystore schemas in TyphonML textual form;
- corresponding query transformation rules, illustrating the way the related TyphonQL queries are transformed.

3.2.1 SMOs on Entity

We present below the transformation rules related to the schema modification operators applied to TyphonML entities.

Entity Add

- *Schema modification operator*: add entity E
- *Schema modification illustration*:

```
1 entity E2 {
2    Attr1: String
3 }
4 changeOperators[add entity E1]
```

Listing 1: Source schema

```
1 entity E2 {
2    Attr1: String
3 }
4 entity E1 {
5 }
```

Listing 2: Target schema

- *Query transformation*: Since, by definition, the new entity $E1$ is not used by any of the existing input queries, the latter remain unchanged.

Entity Remove

- *Schema modification operator*: remove entity E
- *Schema modification illustration*:

```
1 entity E1 {
2    Attr1: String
3 }
4 entity E2 {
5    ...
6 }
7 changeOperators[remove entity E1]
```

Listing 3: Source schema

```
1 entity E2 {
2    ...
3 }
```

Listing 4: Target schema

- *Query transformation*: An existing entity $E1$ with an attribute $Attr1$ is removed from the polystore schema. Consequently, all the TyphonQL queries referencing entity $E1$ have become invalid. Let us consider the following input TyphonQL queries, manipulating instances of entity $E1$:

```
1 from E1 e select e.Attr1,
2 insert E1 {Attr1: "dummy"},
3 delete E1 e where e.Attr1 == "dummy",
4 update E1 e where e.Attr1 == "dummy" set {Attr1: "foobuzz"}
```

The tool identifies these queries as *broken* and generates a corresponding warning in order to inform the user that entity E1 has been removed, making the queries invalid.

```
1 BROKEN
2 #@ Entity E1 removed. This query is broken @#
3 from E1 e select e.Attr1,
4
5 BROKEN
6 #@ Entity E1 removed. This query is broken @#
7 insert E1 {Attr1: "dummy"},
8
9 BROKEN
```

```
10 #@ Entity E1 removed. This query is broken @#
11 delete E1 e where e.Attr1 == "dummy",
12
13 BROKEN
14 #@ Entity E1 removed. This query is broken @#
15 update E1 e where e.Attr1 == "dummy" set {Attr1: "foobuzz"}
```

Entity Rename

- *Schema modification operator*: rename entity $E1$ as $E2$
- *Schema modification illustration*:

```
1 entity E1 {
2    Attr1: String
3 }
4 changeOperators[rename entity E1 as E2]
```

Listing 5: Source schema

```
1 entity E2 {
2    Attr1: String
3 }
```

Listing 6: Target schema

- *Query transformation*: Let us consider the following input TyphonQL queries:

```
1 from E1 e select e.Attr1,
2 insert E1 {Attr1: "dummy"},
3 delete E1 e where e.Attr1 == "dummy",
4 update E1 e where e.Attr1 == "dummy" set {Attr1: "foobuzz"}
```

The Query Migration tool identifies all the TyphonQL queries referencing the old entity name $E1$ and adapt those queries by replacing this name with the new entity name $E2$.

```
1 MODIFIED
2 from E2 e select e.Attr1,
3
4 MODIFIED
5 insert E2 {Attr1: "dummy"},
6
7 MODIFIED
8 delete E2 e where e.Attr1 == "dummy",
9
10 MODIFIED
11 update E2 e where e.Attr1 == "dummy" set {Attr1: "foobuzz"}
```

Entity Merge

- *Schema modification operator*: merge entities $E1$ $E2$ as $EMerged$
- *Schema modification illustration*:

```
1 entity E1 {
2    Attr1: String
3 }
4 entity E2 {
5    Attr2: String
6 }
7 changeOperators[merge entities E1 E2 as EMerged]
```

Listing 7: Source schema

```
1 entity EMerged {
2    Attr1: String
3    Attr2: String
4 }
```

Listing 8: Target schema

- *Query transformation*: When two entities $E1$ and $E2$ are merged into a new entity $EMerged$, the Query Migration tool considers all TyphonQL queries related to at least one of those entities.

```
1 from E1 e1 select e1.Attr1,
2 from E1 e1, E2 e2 select e2.Attr2 where e1.relation1 == e2, e2.Attr2 == "dummy",
3 insert E1 {Attr1: "dummy"},
4 delete E1 e where e.Attr1 == "dummy",
5 update E1 e where e.Attr1 == "dummy" set {Attr1: "foobuzz"}
```

Some queries can be transformed, with a warning indicating that the behavior of the output query may differ from the behavior input query. Some other queries, like insert queries, have become invalid and are marked as *broken*.

```
WARNING
#@ Query return a different QuerySet : EMerged contains attributes from E1 and E2 @#
from EMerged e1 select e1.Attr1,

WARNING
#@ Query return a different QuerySet : EMerged contains attributes from E1 and E2 @#
from EMerged e1 select e1.Attr2 where e1.Attr2 == "dummy",

BROKEN
#@ E1 and E2 merged. @#
insert EMerged {Attr1: "dummy"},

WARNING
#@ E1 and E2 merged. Delete will erase more information than before @#
delete EMerged e where e.Attr1 == "dummy",

MODIFIED
#@ E1 and E2 merged. @#
update EMerged e where e.Attr1 == "dummy" set {Attr1: "foobuzz"}
```

Entity Split

- *Schema modification operator*: split entity vertical E to $E2$ attributes:["$E.B$"]
- *Schema modification illustration*:

```
entity E {
   A : String
   B : String
}
changeOperators
   [split entity vertical E to E2 attributes:["E.B"]]
```

Listing 9: Source schema

```
entity E {
   A : String
}
entity E2 {
   B : String
   to_E -> E [1]
}
```

Listing 10: Target schema

- *Query transformation*: Let us consider the following TyphonQL input queries, manipulating instances of entity E, that has been split into two entities E and $E2$ and a new relationship `to_E` between them. Attribute A remains an attribute of entity E, while attribute B is now an attribute of entity $E2$.

```
from E e select e,
from E e select e.A,
from E e select e.B,
insert E {A: "dummy", B: "dummy"},
delete E e where e.A == "dummy",
update E e where e.A == "dummy" set {A: "foobuzz"}
```

Select queries manipulating entity E can generally be transformed, but a warning is generated to indicate that the output query may return a different result set. In the more simple cases, this transformation consists of renaming the entity name E into $E2$, depending on the attributes manipulated by the query. In more complex cases, the output query relies on a join query between the new entities $E1$ and $E2$, based on relationship `to_E`. In contrast, insert, delete and update queries are generally considered as *broken*, since in all cases, they cannot be replaced with a *single* equivalent query expressed on top of the target schema.

```
WARNING
#@ Entity E split into E, E2 @#
from E e, E2 e2 select e, e2 where e2.to_E == e,
```

```
4
5  WARNING
6  #@ Entity E split into E, E2 @#
7  from E e select e.A,
8
9  WARNING
10 #@ Entity E split into E, E2 @#
11 from E2 e2 select e2.B,
12
13 BROKEN
14 #@ Entity E split into E, E2 @#
15 insert E {A: "dummy", B: "dummy"},
16
17 BROKEN
18 #@ Entity E split into E, E2 @#
19 delete E e where e.A == "dummy",
20
21 BROKEN
22 #@ Entity E split into E, E2 @#
23 update E e where e.A == "dummy" set {A: "foobuzz"}
```

Entity Migrate

- *Schema modification operator*: migrate $E1$ to $dbName$
- *Schema modification illustration*:

```
1  entity E1 {
2     Attr1 : Date
3     Attr2 : int
4  }
5  relationaldb relationDB {
6     tables {
7       table{
8         E1Table: E1
9        }
10      }
11 }
12 documentdb DocumentDatabase {
13 }
14 changeOperators [migrate E1 to DocumentDatabase]
```

Listing 11: Source schema

```
1  entity E1 {
2     Attr1 : Date
3     Attr2 : int
4  }
5  relationaldb relationDB {
6  }
7  documentdb DocumentDatabase {
8     collections {
9        E1Collections: E1
10    }
11 }
```

Listing 12: Target schema

- *Query transformation*: The migrate operator enables the migration of data structure and instances of an entity E to another database platform, in the above example, from a relational table to a MongoDB collection. All TyphonQL queries that access entity $E1$ remain valid. They are therefore left unchanged by the Query Migration tool.

3.2.2 SMOs on Attribute

In this section, we present the query transformation rules corresponding to the change operators that relate to TyphonML attributes.

Attribute Add

- *Schema modification operator*: add attribute $A : AType$ to E

- *Schema modification illustration*:

```
entity E1 {
   Attr1: String
}
changeOperators[add attribute Attr2 : String to E1]
```

Listing 13: Source schema

```
entity E1 {
   Attr1: String
   Attr2: String
}
```

Listing 14: Target schema

- *Query transformation*: Let us consider the following input TyphonQL queries, manipulating entity $E1$:

```
from E1 e select e.Attr1,
insert E1 {Attr1: 10},
delete E1 e where e.Attr1 == 15,
update E1 e where e.Attr1 == 10 set {Attr1: 15}
```

Since a new attribute $Attr2$ has been added to entity $E1$, some select queries may return a different result set, hence a corresponding warning. Update queries may remain unchanged, insert queries are considered as *broken* due to the missing attribue value, while delete queries remain valid but require a warning to the user.

```
WARNING
#@ Attribute Attr2 added to E1. Result of the query may have changed @#
from E1 e select e.Attr1,

BROKEN
#@ Attribute Attr2 added to E1. Insert is broken @#
insert E1 {Attr1: 10},

WARNING
#@ Attribute Attr2 added to E1. You may delete more information than expected @#
delete E1 e where e.Attr1 == 15,

UNCHANGED
update E1 e where e.Attr1 == 10 set {Attr1: 15}
```

Attribute Remove

- *Schema modification operator*: remove attribute "$E.A$"
- *Schema modification illustration*

```
entity E1 {
   Attr1 : String
   Attr2 : int
}
changeOperators[remove attribute "E1.Attr2"]
```

Listing 15: Source schema

```
entity E1 {
   Attr1: String
}
```

Listing 16: Target schema

- *Query transformation*: Let us consider the following input TyphonQL queries, manipulating instances of the $E1$ entity:

```
from E1 e select e.Attr2,
insert E1 {Attr2: 10},
delete E1 e where e.Attr2 == 15,
update E1 e where e.Attr2 == 10 set {Attr1: 10},
from E1 e select e where e.Attr1 == "test",
update E1 e where e.Attr1 == "dummy" set {Attr1: "test"}
```

Since attribute $Attr2$ has been removed from entity $E1$ all TyphonQL queries explicitly referencing $Attr2$ are no longer valid, and are reported as *broken*. Other queries may remain unchanged but require a warning indicating that they might differ from the input queries in terms of their result set or execution impact.

```
1  BROKEN
2  #@ Attribute E1.Attr2 removed @#
3  from E1 e select e.Attr2,
4
5  BROKEN
6  #@ Attribute E1.Attr2 removed @#
7  insert E1 {Attr2: 10},
8
9  BROKEN
10 #@ Attribute E1.Attr2 removed @#
11 delete E1 e where e.Attr2 == 15,
12
13 BROKEN
14 #@ Attribute E1.Attr2 removed @#
15 update E1 e where e.Attr2 == 10 set {Attr1: 10},
16
17 WARNING
18 #@ Query result might differ : Attribute E1.Attr2 removed @#
19 from E1 e select e where e.Attr1 == "test",
20
21 WARNING
22 #@ Query result might differ : Attribute E1.Attr2 removed @#
23 update E1 e where e.Attr1 == "dummy" set {Attr1: "test"}
```

Attribute Rename

- *Schema modification operator*: rename attribute "$E.A$" to $NewName$
- *Schema modification illustration*:

```
1 entity E1 {
2    Attr1 : String
3    Attr2 : int
4 }
5 changeOperators
6   [rename attribute "E1.Attr1" to NewAttributeName]
```

Listing 17: Source schema

```
1 entity E1 {
2    NewAttributeName: String
3    Attr2 : int
4 }
```

Listing 18: Target schema

- *Query transformation*: The Query Migration tool identifies all TyphonQL queries referencing the old attribute name $Attr1$, as it is the case for the following input queries:

```
1 from E1 e select e.Attr1,
2 insert E1 {Attr1: "dummy"},
3 delete E1 e where e.Attr1 == "dummy",
4 update E1 e where e.Attr1 == "dummy" set {Attr1: "foobuzz"}
```

The tool then renames attribute "E1.Attr1" as NewAttributeName in all those queries, as follows:

```
1  MODIFIED
2  from E1 e select e.NewAttributeName,
3
4  MODIFIED
5  insert E1 {NewAttributeName: "dummy"},
6
7  MODIFIED
8  delete E1 e where e.NewAttributeName == "dummy",
9
10 MODIFIED
11 update E1 e where e.NewAttributeName == "dummy" set {NewAttributeName: "foobuzz"}
```

Attribute Change Type

- *Schema modification operator*: change attribute "$E.A$" type $NewType$

- *Schema modification illustration*

```
entity E1 {
   Attr1 : String
   Attr2 : int
}
changeOperators[change attribute "E1.Attr1" type Date]
```

Listing 19: Source schema

```
entity E1 {
   Attr1 : Date
   Attr2 : int
}
```

Listing 20: Target schema

- *Query transformation*: The Query Migration tool identifies all the TyphonQL queries explicitly referencing the updated attribute $Attr1$.

```
from E1 e select e.Attr1,
insert E1 {Attr1: "dummy"},
delete E1 e where e.Attr1 == "dummy",
update E1 e where e.Attr1 == "dummy" set {Attr1: "foobuzz"},
from E1 e select e where e.Attr2 > 10
```

The general structure of those TyphonQL queries remains valid, but a warning is provided in order to remind the user to check the conformance of the input values used for the $Attr1$ attribute, as well as the type of the related host program variables.

```
WARNING
#@ The type of the attribute Attr1 from E1 change @#
from E1 e select e.Attr1,

WARNING
#@ The type of the attribute Attr1 from E1 change @#
insert E1 {Attr1: "dummy"},

WARNING
#@ The type of the attribute Attr1 from E1 change @#
delete E1 e where e.Attr1 == "dummy",

WARNING
#@ The type of the attribute Attr1 from E1 change @#
update E1 e where e.Attr1 == "dummy" set {Attr1: "foobuzz"},

WARNING
#@ The type of the attribute Attr1 from E1 change @#
from E1 e select e where e.Attr2 > 10
```

3.2.3 SMOs on Relationship

In this section, we present the query transformation rules corresponding to schema modification operators on TyphonML relationships.

Relationship Add

- *Schema modification operator*: add relation $relName$ to $E1$ -> $E2$
- *Schema modification illustration*:

```
entity E1 {
   Attr1 : String
   relation1 -> E2
}
entity E2 {
   Attr2 : String
}
changeOperators[add relation relation2 to E1 -> E2]
```

Listing 21: Source schema

```
entity E1 {
   Attr1 : String
   relation1 -> E2
   relation2 -> E2
}
entity E2 {
   Attr2 : String
}
```

Listing 22: Target schema

- *Query transformation*: Let us consider the following input TyphonQL queries:

```
1 from E1 e1 select e1.relation1,
2 from E1 e1 select e1 where e1.relation1,
3 from E1 e1 select e1 where e1.relation1.Attr2 == "dummy"
```

Those input queries are left unchanged by the Query Migration tool, since they remain valid despite the new relation $relation2$ that has been added between $E1$ and $E2$.

```
1 UNCHANGED
2 from E1 e1 select e1.relation1,
3
4 UNCHANGED
5 from E1 e1 select e1 where e1.relation1,
6
7 UNCHANGED
8 from E1 e1 select e1 where e1.relation1.Attr2 == "dummy"
```

Relationship remove

- *Schema modification operator*: remove relation "$E.relName$"
- *Schema modification illustration*:

```
1 entity E1 {
2    Attr1 : String
3    relation1 -> E2
4 }
5 entity E2 {
6    Attr2 : String
7 }
8 changeOperators[remove relation "E1.relation1"]
```

Listing 23: Source schema

```
1 entity E1 {
2    Attr1 : String
3 }
4 entity E2 {
5    Attr2 : String
6 }
```

Listing 24: Target schema

- *Query transformation*: Relation $Attr1$ has been removed from entity $E1$. Let us consider the following input TyphonQL queries, which explicitly refer to the deleted relation:

```
1 from E1 e1 select e1.relation1,
2 from E1 e1 select e1 where e1.relation1,
3 from E1 e1 select e1 where e1.relation1.Attr2 == "dummy"
```

All those TyphonQL queries have become invalid, and are therefore considered as *broken* by the Query Migration tool:

```
1 BROKEN
2 #@ The relation relation1 was removed @#
3 from E1 e1 select e1.relation1,
4
5 BROKEN
6 #@ The relation relation1 was removed @#
7 from E1 e1 select e1 where e1.relation1,
8
9 BROKEN
10 #@ The relation relation1 was removed @#
11 from E1 e1 select e1 where e1.relation1.Attr2 == "dummy"
```

Relationship rename

- *Schema modification operator*: rename relation "$E1.relName1$" as $relName2$

- *Schema modification illustration*:

```
entity E1 {
   Attr1 : String
   relation1 -> E2
}
entity E2 {
   Attr2 : String
}
changeOperators
    [rename relation "E1.relation1" as relation2]
```

Listing 25: Source schema

```
entity E1 {
   Attr1 : String
   relation2 -> E2
}
entity E2 {
   Attr2 : String
}
```

Listing 26: Target schema

- *Query transformation*: The Query Migration tool identifies all the TyphonQL queries referencing the old relationship name $relation1$, as it is the case for the following queries:

```
from E1 e1 select e1.relation1,
from E1 e1 select e1 where e1.relation1,
from E1 e1 select e1 where e1.relation1.Attr2 == "dummy"
```

  The tool then adapts those queries by replacing the outdated relationship name $relation1$ with the new relationship name $relation2$.

```
MODIFIED
from E1 e1 select e1.relation2,

MODIFIED
from E1 e1 select e1 where e1.relation2,

MODIFIED
from E1 e1 select e1 where e1.relation2.Attr2 == "dummy"
```

Relationship change cardinality

- *Schema modification operator*: change cardinality "$E.relName$" as $relCardinality$
- *Schema modification illustration*:

```
entity E1 {
   Attr1 : String
   relation1 -> E2 [*]
}
entity E2 {
   Attr2 : String
}
changeOperators
  [change cardinality "E1.relation1" as 0..*]
```

Listing 27: Source schema

```
entity E1 {
   Attr1 : String
   relation1 -> E2 [0..*]
}
entity E2 {
   Attr2 : String
}
```

Listing 28: Target schema

- *Query transformation*: The cardinality of the relation between the entities $E1$ and $E2$ has been modified. Let us consider the following input TyphonQL queries:

```
from E1 e1 select e1.relation1,
from E1 e1 select e1 where e1.relation1,
from E1 e1 select e1 where e1.relation1.Attr2 == "dummy",
delete E1 e where e.relation1 == "5"
```

  The structure of those queries remain valid, but the user is warned that they queries may now have a different behavior.

```
WARNING
#@ Cardinality of relation relation1 as changed to 0..* @#
from E1 e1 select e1.relation1,
```

```
4
5 WARNING
6 #@ Cardinality of relation relation1 as changed to 0..* @#
7 from E1 e1 select e1 where e1.relation1,
8
9 WARNING
10 #@ Cardinality of relation relation1 as changed to 0..* @#
11 from E1 e1 select e1 where e1.relation1.Attr2 == "dummy",
12
13 WARNING
14 \#@ Cardinality of relation relation1 as changed to 0..* @\#
15 delete E1 e where e.relation1 == "5"
```

4 Example Query Migration Scenario

In this section, we further illustrate the use of the query migration tool by considering a schema evolution scenario involving a *chain* of schema modification operators. Let us assume the source and target TyphonML schemas depicted respectively on the left and on the right of Figure 2.

Let us now consider the following input TyphonQL queries expressed on top of the source TyphonML schema:

```
1  from User u, CreditCard c select c where u.paymentsDetails == c, u.name == "Doe",
2
3  insert User {id:5, name:"John", surname:"Doe"},
4
5  from Review r, Product p select r where p.review == r, p.id == "145",
6
7  from Order o select o where o.date > 1998,
8
9  update Order o where o.totalAmount > 99999 set {},
10
11 delete Comment c where c.id == "122"
```

The query migration tool, applied to this set of queries under this schema evolution scenario would produce the following annotated output queries:

```
1  WARNING
2  #@ Query return a different QuerySet : User contains attributes from User and CreditCard @#
3  from User u select u where u.name == "Doe"  ,
4
5  BROKEN
6  #@ User and CreditCard merged @#
7  insert User {id:5, name:"John", surname:"Doe"},
8
9  WARNING
10 #@ Attribute rating added to Review. Result of the query may have changed @#
11 from Review r, Product p select r where p.review == r, p.id == "145",
12
13 BROKEN
14 #@ Attribute Order.date removed @#
15 from Order o select o where o.date > 1998,
16
17 WARNING
18 #@ Query result might differ : Attribute Order.date removed @#
19 #@ The type of the attribute totalAmount from Order changed @#
20 update Order o where o.totalAmount > 99999 set {},
21
22 UNCHANGED
23 delete Comment c where c.id == "122"
```

We see that some of the output queries are annotated with several warnings, due to the successive application of several transformation rules.

```
1  entity Review {
2      id : String
3      product -> Product[1]
4  }
5  entity Product {
6      id : String
7      name : String
8      description : String
9      photo : Blob
10     review :-> Review[0..*]
11     orders -> Product[0..*]
12 }
13 entity Order {
14     id : String
15     date : Date
16     totalAmount : int
17
18     products -> Product.products[0..*]
19     users -> User[1]
20     paidWith -> CreditCard[1]
21 }
22 entity User {
23     id : String
24     name : String
25     surname : String
26     comments :-> Comment[0..*]
27     paymentsDetails :-> CreditCard[0..*]
28     orders -> Order[0..*]
29 }
30 entity Comment {
31     id : String
32     responses :-> Comment[0..*]
33 }
34 entity CreditCard {
35     id : String
36     number : String
37     expiryDate : Date
38 }
39
40 relationaldb RelationalDatabase {
41     tables {
42         table {
43             Order : Order
44             index orderIndex {
45                 attributes ("Order.date")
46             }
47             idSpec ("Order.date")
48         }
49         table {
50             User : User
51             index  userIndex {
52                 attributes ('User.name')
53                 }
54             idSpec ('User.name')
55         }
56         table {
57             CreditCard : CreditCard
58             idSpec ("CreditCard.number")
59         }
60     }
61 }
62 changeOperators [
63   add attribute rating : int to Review,
64   add relation responses to Review -> Comment[0..*],
65   merge entities User CreditCard as User,
66   remove attribute "Order.date",
67   change attribute "Order.totalAmount" type Real
68 ]
```

```
1  entity Review {
2      id : String
3      rating: int
4      product -> Product[1]
5      responses -> Comment[0..*]
6  }
7  entity Product {
8      id : String
9      name : String
10     description : String
11     photo : Blob
12     review :-> Review[0..*]
13     orders -> Product[0..*]
14 }
15 entity Order {
16     id : String
17     totalAmount : Real
18
19     products -> Product.products[0..*]
20     users -> User[1]
21     paidWith -> User[1]
22 }
23 entity User {
24     id : String
25     name : String
26     surname : String
27     number : String
28     expiryDate : Date
29
30     comments :-> Comment[0..*]
31     orders -> Order[0..*]
32 }
33 entity Comment {
34     id : String
35     responses :-> Comment[0..*]
36 }
37
38 relationaldb RelationalDatabase {
39     tables {
40         table {
41             Order : Order
42             index orderIndex {
43                 attributes ("Order.date")
44             }
45             idSpec ("Order.date")
46         }
47         table {
48             User : User
49             index  userIndex {
50                 attributes ('User.name')
51                 }
52             idSpec ('User.name')
53         }
54         table {
55             CreditCard : CreditCard
56             idSpec ("CreditCard.number")
57         }
58     }
59 }
```

Figure 2: Source (left) and target (right) TyphonML schemas.

5 Implementation

In this section, we elaborate on some implementation details of our *Query Migration Tool*. We first present the main architecture of the tool (Section 5.1). Then, we describe the way to use this tool through a brief user guide (Section 5.2).

5.1 Tool Description

The main architecture of the query migration tool is built on *Rascal* language which is a domain-specific language for metaprogramming that is used to solve problems in the domain of source code analysis and transformation [3]. Rascal is also used in Work Package 4 to implement the TyphonQL language compilers. We could therefore benefit from existing grammars of (1) the TyphonQL language and (2) the TyphonML XMI format.

The query migration tool is released as an eclipse plugin. Using this plugin, the user can create an input file (called *evolution script*) that includes:

- a link to the source TyphonML schema;
- the schema modification operator(s) applied to this source schema;
- all the TyphonQL queries, expressed on top of the source schema, that should be transformed according to thoses SMO(s).

The query migration tool iterates on the list of schema modification operators and, for each of them, successively transform and/or annotate the TyphonQL queries given as input.

5.2 User Guide

This section provides a brief, illustrated user guide of the query migration tool allowing user to migrate existing TyphonQL queries to an evolving TyphonML schema.

Everything starts from the structure view of the Eclipse project, as shown in Figure 3.

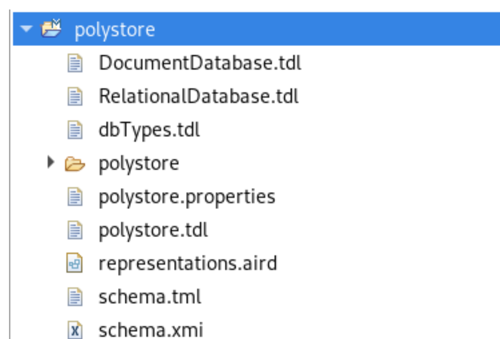

Figure 3: View of the Eclipse project structure

The file *schema.xmi* contains the source TyphonML schema as well as the change operators that are generated from the TyphonML editor. In order to migrate related TyphonQL queries accordingly, a new query evolution

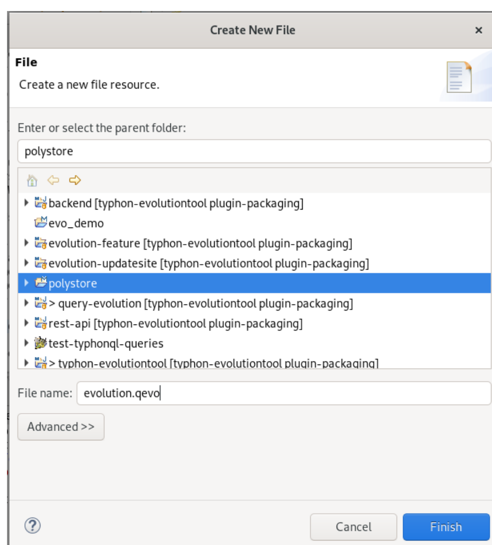

Figure 4: Creation of a new *.qevo* file

script file should be created. This is done as shown in Figure 4. The file extension of a query evolution script is *.qevo*.

Initially, the created evolution script consists of an empty file. Then it should be filled, as presented in Figure 5. Basically, the script is made of three parts :

- **Import**: the path to the XMI file corresponding to the source TyphonML schema;
- **Change operators**: the (set of) change operator(s) applied to the source TyphonML schema. These operators can simply be copy-pasted from the *.tml* form of the source TyphonML schema[1];
- **Queries**: the set of input TyphonQL queries, separated by a comma.

```
1  import schema.xmi;
2
3  changeOperators[
4      merge entities E1 E2 as EMerged
5  ]
6
7  insert E1 {Attr1: "dummy"},
8
9  from E1 e1, E2 e2 select e2 where e1.relation1 == e2, e1.Attr2 == "dummy"  ,
10
11 update E1 e where e.Attr1 == "dummy" set {Attr1: "foobuzz"},
12
13 from E3 e select e
```

Figure 5: Example query evolution script.

Once the script is fully written, the query migration process can be triggered through a simple **right click** on the evolution script. A menu appears, one can then select **TyphonEvolution** -> **evolve**, as shown in Figure 6.

The queries written by the user in the evolution script are then automatically updated and annotated in the same *.qevo* file (Figure 7). In this example, we can see all the possible output cases of a query migration:

[1]Note that once the query migration will be integrated in the polystore platform, the change operators will be automatically extracted from the *.xmi* file and will not be required from the user anymore.

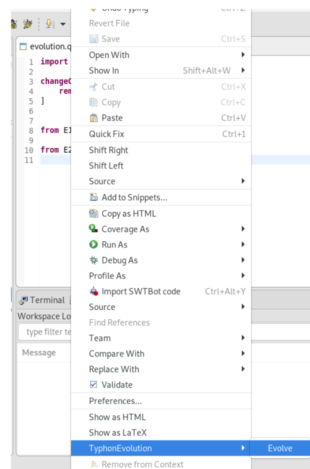

Figure 6: Running the query migration tool.

- The first output query (lines 7-9) is set as *broken* since it inserts data in an entity *E1*, which has been merged with an another entity into a new entity *EMerged*. Therefore the user is warned that the query should be modified manually.
- The second query (lines 11-13) is annotated with a warning. Indeed, the user is requesting data from entity $E1$. Since the latter was merged with entity $E2$, the user is informed about the impact of this change on the result set of the query. Indeed, the new query result now returns information from *E1* and *E2*.
- The third query (lines 15-17) was successfully transformed into an equivalent query. The behavior of the output query is the same as the input query.
- The fourth query (lines 19-20): can remain unchanged, since it is not impacted the merge operation.

The output annotations help the user to identify which queries became invalid and provide useful information in order to further adapt the code querying the polystore to the target TyphonML schema.

```
1  import schema.xmi;
2
3  changeOperators[
4      merge entities E1 E2 as EMerged
5  ]
6
7  BROKEN
8  #@ E1 and E2 merged. @#
9  insert EMerged {Attr1: "dummy"},
10
11 WARNING
12 #@ Query return a different QuerySet : EMerged contains attributes from E1 and E2 @#
13 from EMerged e1 select e1 where e1.Attr2 == "dummy",
14
15 MODIFIED
16 #@ E1 and E2 merged. @#
17 update EMerged e where e.Attr1 == "dummy" set {Attr1: "foobuzz"},
18
19 UNCHANGED
20 from E3 e select e
```

Figure 7: Output queries.

6 Conclusions

In this deliverable, we have presented a tool-supported method for TyphonQL query migration under polystore schema evolution. This method and tool support the automated adaptation of existing TyphonQL queries to an evolving TyphonML schema. More specifically, we have described the way the TyphonQL queries are transformed in accordance to schema modification operators applied to a source TyphonML schema, in order (1) to produce (almost) equivalent output queries expressed on top of the target TyphonML model or (2) to identify the queries that became invalid and could not be migrated. Four categories of output queries are returned by the query migration tool : (i) Unchanged queries, that remain valid; (ii) Modified queries, that have been transformed and are equivalent; (iii) Modified with warning(q), that are valid queries but may expose a different behavior; or (iv) Broken queries, that became invalid and cannot be migrated towards the target schema.

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

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] P. Klint, T. van der Storm, and J.J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *In Proceedings of the 9th International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, pages 168–177. IEEE Computer Society, 2009.

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

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

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