Project Number 780251

D4.5 TyphonQL Compilers and Interpeters (Final Version)

Version 1.0
8 July 2020
Final

Public Distribution

Centrum Wiskunde & Informatica (CWI) and SWAT.engineering

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**

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 SÃ¡nchez GonzÃ¡lez 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
0.3	First full draft	29 June 2020
0.8	Version for partner reviews	6 July 2020
1.0	Final updates from QA review	8 July 2020

Table of Contents

Executive Summary

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

TyphonQL is a new query language that strictly operates at the level of the conceptual data model developed in Work Package 2. Queries expressed in this language are partitioned over different database back-ends and the results of those partial native queries are recombined to obtain the end result of the query.

This document presents the final version of the TyphonQL compilers and interpreters that perform the translation of TyphonQL queries to native queries on top of SQL databases, MongoDB document stores, Cassandra Key-Value stores, and Neo4J graph databases. Furthermore, this deliverable details the TyphonQL aggregation framework, the TyphonQL type checker, and the IDE.

1 Introduction

This deliverable reports on the final version of the TyphonQL compilers and interpreters. More specifically, it is complementary to earlier deliverables of Work Package 4 that presented initial and interim versions of the language. TyphonQL is based on the language design as reported in D4.2 [2], the initial prototype documented in D4.3 [3], and the interim version reported on in D4.4 [11]. Work package 4 is also concerned with developing the Data Access Layer Generator; this work is documented in a separate deliverable, D4.6 [12].

The code of TyphonQL can be found at `https://github.com/typhon-project/typhonql`.

1.1 Structure of the Deliverable

Section 2 briefly summarizes the architecture of the TyphonQL compiler and status in terms of feature support as described in D4.4 [11]. This will provide the necessary background to the additional features described here, that together make up the final version of the language and the compiler.

More specifically, Section 3 identifies core challenges and achievements of WP4. Section 3.1 provides additional detail on a core aspect of the TyphonQL compiler: how queries are partitioned over different back-ends, using heuristics to obtain efficient execution. The fact that TyphonQL targets n potential back-ends is elaborated upon in Section 3.2, highlighting the combinatorial effect of relations between entities. Finally, Section 3.3 provides initial insight into the overhead of the TyphonQL compiler compared to query execution time.

Section 4 provides a complete overview of the basic primitive data types supported by TyphonQL. Additional detail will be provided on the new data types `blob` and the Geo data types `point` and `polygon`. TyphonML [7] supports the definition of custom data types, to group together sets of primitive typed fields in reusable abstractions. In Section 4.2 we describe how custom data types are supported in TyphonQL through normalization.

Section 5 details the way TyphonQL supports aggregation features. This includes the usual group-by, limit, order by, and having clauses well-known from SQL, as well as common aggregation functions (count, sum, max, etc.). The design of TyphonQL aggregation is orthogonal to the usual partitioning, since aggregation does not distribute (algebraicly) over intermediate results.

Next to MongoDB and MariaDB, the TyphonQL compiler currently supports two additional back-ends: Cassandra for key-value stores, and Neo4J for graph databases. Section 6 provides the details on how support was realized.

TyphonQL is a language and compiler for querying hybrid polystores, but additionally comes with tools to help TyphonQL users and/or developers to experiment with TyphonQL or develop third-party tools. Section 7 introduces the TyphonQL IDE (Section 7.1), the type checker (Section 7.2), and the TyphonQL AST framework (Section 7.3). The latter can be used to process TyphonQL ASTs in Java using dedicated Visitors; it is currently used by the Typhon Analytics Framework (WP5 [10]).

This deliverable concludes by evaluating the language and compiler of TyphonQL with respect to the use case requirements stated in D1.1 [15] in Section 8, and concluding remarks in Section 9.

The full Rascal [6] syntax definition of TyphonQL can be found in Appendix B.

```
from Person p, Review r select r.text, p.name
  where p.name == "Pablo", p.reviews == r
```

```
script([
    step(
      "Inventory",
      sql(executeQuery(“Inventory",
        "select `p`.`Person.name` as `p.Person.name`,
                `p`.`Person.@id` as `p.Person.@id`,
                `junction_reviews$0`.`Review.user` as `p.Person.reviews`
         from `Person` as `p`,
              `Person.reviews-Review.user` as `junction_reviews$0`
         where (`p`.`Person.name`) = (\'Pablo\')
           and (`junction_reviews$0`.`Person.reviews`)
             = (`p`.`Person.@id`);")),
      ()),
    step(
      "Reviews",
      mongo(find(“Reviews","Review",
        "{\"_id\": ${p_reviews_0}}","{\"text\": 1}")),
      ("p_reviews_0":field("Inventory","p","Person","reviews")))
  ])
```

Figure 1: Compiling a TyphonQL query to a script of native query invocations

2 Overview of TyphonQL Compiler as per D4.4

To provide enough background in order to make this deliverable self-contained, this section briefly summarizes the key points of earlier deliverables of Work Package 4. The following sections can then be interpreted as updates and additions to the work discussed earlier.

TyphonQL is a query and data manipulation language (DML) with a syntax similar to SQL, but a semantics that is object-oriented, in the sense that the structures that can be queried are not tables (or documents, or arrays, or graphs,... etc.), but rather graph structures consisting of objects and relations between them, typed by a class-based schema defined in TyphonML [7].

TyphonQL queries are compiled to native query languages, such as SQL, Cypher, CQL, or MongoDB API calls. Since a TyphonQL query can span multiple back-ends, a partitioning process is implemented which slices a query in to n separate queries for potentially n back-ends, which are then individually compiled to native back-end query languages. Details about ordered partitioning are provided in D4.4 [11] ans Section 3.1.

The results are combined using an iteration architecture that retrieves intermediate results from a back-end, passes relevant information into the native queries of the next back-end through prepared statements, and so on, until after the last back-end has been hit, the final result can be constructed. The design of this architecture is detailed in Appendix A.

TyphonQL DML statements (insert/update/delete) are always strictly local to a single back-end, *except* for managing referential integrity. For instance, deleting an entity requires deleting all the *contained* entities (as per the TyphonML model), which may live on another back-end. Similarly, inserting or updating a reference of an entity might incur updating its inverse pointer on another back-end.

The unifying abstraction here is an internal *script* language, which is executed by the TyphonQL engine implemented in Java. Figure 1 shows an example of how a single TyphonQL query is compiled to a script containing multiple steps, to be executed on native back-ends. This kind of script can thus be seen as an abstract machine for TyphonQL, where the primitive instructions are query and DML statement calls to various back-ends.

Before executing a TyphonQL query or statement, the following normalizations and desugarings are performed to simplify the development of the compiler:

- Expansion of lone variables[1] in result to all attributes [11]
- NLP `freetext` types are lifted into their own entity, declaring the attributes corresponding to the NLP tasks enabled [11].
- Path expansion of object navigation into explicit **`where`**-clauses [11]
- Inline custom data types (Section 4.2)
- Hoist key-value store attributes (Section 6.1)
- Slice a query into filtering and aggregation slices if aggregation features are used (Section 5).

Other components of Work Package 4 that have been reported earlier on, and will not be repeated here, include:

- The design and overview of the TyphonQL syntax [2]
- Basic query and DML compilation [3]
- The mapping of TyphonML models to native schemas [3]
- Implementation of evolution operators [3] to support Work Package 6 [8, 9]
- Integration with the Polystore environment [11]

3 Challenges and Achievements

TyphonQL is a core component of the Typhon project where a lot of heavy lifting takes place, to obtain a query engine that is rich in features, performant, and open for extension. Below we survey key challenges and achievements of realizing TyphonQL.

3.1 Ordered Partitioning

Partitioning refers to the process over different back-ends to evaluate slices of the original query using native capabilities of database back-ends. The individual results are then combined in the final result of the query. The recombination of intermediate results is detailed in Appendix A.

Consider the example query of Figure 1:

```
from Person p, Review r select r.text, p.name
  where p.name == "Pablo", p.reviews == r
```

In this case, `Person` entities are stored on a MariaDB database, and `Review` entities on MongoDB.

Partitioning can be performed in multiple ways. As a heuristic, the TyphonQL compiler orders potential partitionings based on the *filter weight* on entities mapped to a certain back-end: the most constrained query slices are executed first, so that the intermediate result sets are likely to be the smallest.

[1] A lone variable is a binding of an entity in a select-clause of a query, such as `p` in **`from`** `Person p` **`select`** `p`, indicating that all attributes are requested.

Person (Maria)	Review (Mongo)	Person (Maria)	Review (Mongo)

`from Person p`
`select p.name, p.reviews`
`where p.name == "Pablo"`
			`from Review r`
			`select r.text, r.@id`

	`from Review r`
	`select r.text`
	`where r.@id == ??`
		`from Person p`
		`select p.name`
		`where p.name == "Pablo",`
		`  p.reviews == ??`	

Table 1: Two ways of partitioning the query of Figure 1

This is illustrated in Table 1, which shows two ways of partitioning the above query. The left-hand side shows a partitioning where the MariaDB back-end is hit first: it retrieves all persons named “Pablo” and their reviews. These results are then propagated to the next phase, where the review texts of those persons are retrieved. Note that the where-clause uses a placeholder in the constraint on the reviews identity; this is where data coming from one back-end is passed on to another back-end (cf. Appendix A). In the example, the review identities of persons called “Pablo” are substituted for the placeholder before invoking the native Mongo query command.

The right-hand side of Table 1 shows a partitioning where the MongoDB back-end executes first. It can be seen that this is less efficient (in the average case) than the first partitioning, because in this case first *all* review texts and identities are retrieved, before filtering out the persons who are not called “Pablo”. For this reason, the TyphonQL partitioner will chose the first partitioning. The result is shown in the script of Figure 1, where the first step consists of an SQL query, and the second step represents a call to the MongoDB `find` API.

3.2 Mapping Complexity

One of the key challenges of the TyphonQL compiler is how to bridge the conceptual gap between the high-level model of both TyphonML and TyphonQL to the internal models of the various back-ends. For instance, TyphonML has an object-oriented model inspired from class-based meta models used in model-driven frameworks, such as EMF [1]. SQL databases, however, require an interpretation in the relational model, where data is stored in normalized tables and object-oriented references are encoded as (reverse) foreign keys.

D4.4 [11] provided detail on how such references are normalized to where-clauses to make relational joins explicit; however, for MongoDB such normalization should be partially reverted, since MongoDB is closer to the original object-oriented idea of references (be they bidirectional or not). To complicate things more, however, MongoDB has no native concept of cross-reference, but it does have native concept of containment and collection.

As a result, the TyphonQL compiler has to deal with the cartesian product of combinations between kinds of relations between entities, cardinalities, containment or not, and how back-ends actually realize them. This cartesian product could be characterized as the following:

$$\begin{aligned} &\{mongo, maria, ...\}\\ \times&\{mongo, maria, ...\}\\ \times&\{one, zero\_one, zero\_many, one\_many\}\\ \times&\{containment, xref\}\\ \times&\{insert, delete, update, select\}\\ \times&\{direct, inverse\}\end{aligned}$$

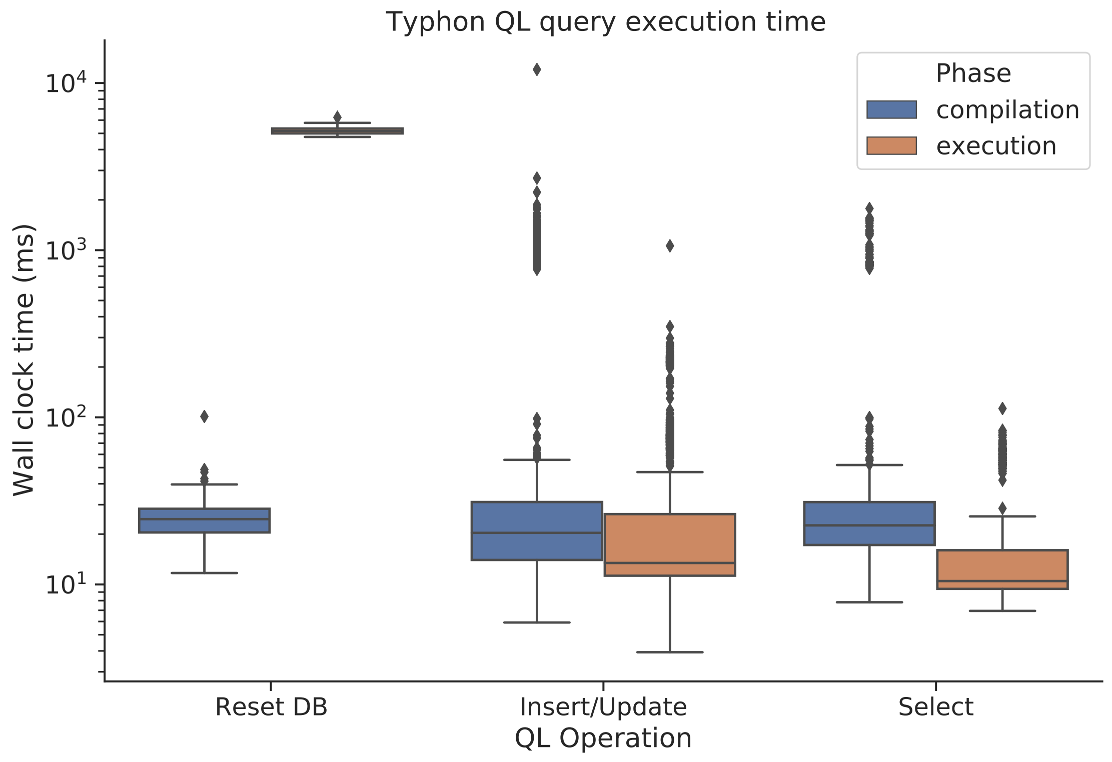


Figure 2: Distribution of running times of `resetDatabase`, `runUpdate`, and `runQuery` during our test-suite (log-scale)

The number of combinations captures the kinds of relations between different back-ends, of different cardinalities, whether they are containment or cross-ref (xref), how they affect the TyphonQL constructs, and whether navigation along them is direct or via an inverse.

The addition of a new back-end, multiplies this product once more. In the current architecture, we have leveraged Rascal's capabilities for modular specification of case-based functions to make this product open for extension: a new back-end, means adding new cases of these functions to cover the new relevant combinations. Without reducing the combinatoric complexity, it becomes more manageable this way.

3.3 Overhead of the Compiler

Since the TyphonQL compiler does quite some work to normalize, partition and translate TyphonQL statements to native expressions for each back-end, it is instructive to assess the overhead of the compiler compared to actual query execution time.

Figure 2 shows results from a micro-benchmark we have run based on the TyphonQL test set. It shows the distribution of running times over 2304 measurements resulting from executing the test suite, divided over three categories:

- Reset DB: all tests run with a clean Polystore; reset database drops all SQL tables, Mongo collections, Cassandra key spaces etc. and reinitializes the structures based on the test TyphonML model.
- DML statement: this measures the times of insert/update/delete statements.

Type	Description	Examples
`int`	32 bit integer	`123, -567, 0`
`bigint`	64 bit integer	`2873498732874387483938490204736 4`
`string[`n`]`	n-bounded string	`“this is a string”`
`text`	arbitrary length text	`“this string can be arbitrary long”`
`bool`	booleans	`true, false`
`float`	IEEE float	`1.323, -123.0e10`
`date`	Date	`$2020-07-3$`
`datetime`	Date and time	`$2020-07-3T23:45:00+2$`
`freetext[`$\bar{x}$`]`	text field with NLP features $\bar{x}$	`“some prose”` (e.g., with Tokenisation enabled in $\bar{x}$)
`blob`	Binary Large OBjects	`#blob:6ca820c8-2104-4f29-b705-1f86fb65fb81`
`point`	Geographical point	`#point(23.4 343.34)`
`polygon`	Closed polygon of line segments	`#polygon((23.4 343.34), (2.0 0.0))`

Table 2: Primitive Data Types supported by TyphonQL

- Select: the compilation and execution time of TyphonQL queries.

As can be seen from the box plots of Figure 2, resetting the database takes the most time, which is to be expected. The overhead of the compiler is relatively small. This is different for compilation and execution of individual statements and queries, where compilation is slightly more expensive than execution. Note, however, that the tests run on a Polystore where the number of records/documents is low. It is to be expected that increased data sizes will make the difference between compilation and execution more pronounced, in that the overhead of compilation will be amortized over the cost of executing the queries.

4 Primitive Data Types

4.1 Basic Primitives

TyphonML, and hence TyphonQL, supports a fixed set of basic data types for the attributes of entities. Primitive data types are also used in the definition of custom data types, which will be discussed below in Section 4.2. The final list of primitive types is shown in Table 2, including example syntax of their literal representation.

Compared to the interim version of TyphonQL (as reported in D4.4 [11]), there are a number of changes. First, 32 bit and 64 bit integers are now distinguished as **`int`** and `bigint`, respectively. Similarly, string valued attributes can now be bound with a maximum size via `string[`n`]`. Arbitrarily sized text attributes may use the `text` type. Furthermore, TyphonQL now supports dates and date-time values using the types **`date`** and `datetime`.

The most important changes to this list are the **`blob`** type (for Binary Large OBjects), and the geo types `point` and `polygon`. The **`blob`** type supports identifying a BLOB resource using the syntax shown in Table 2. It consists of a Universally Unique Identifier (UUID) used as a key in some map containing the binary resource. This resource needs to be provided through the Java or REST API to the TyphonQL engine, which will then replace the **`#blob:`**. . . literals with the actual binary contents when interfacing with native back-ends.

The `point` and `polygon` types represent common geo-spatial data types, supported by many back-ends, including MariaDB and MongoDB. In addition to the new types and literal syntax, these geo-spatial types also come with special operators to compute distances, intersections, and inclusions.

	MariaDB	MongoDB	Cassandra	Neo4J
Arithmetic	●		●	●
Boolean	●	●	●	●
Comparison	●	◐[a]	●	●
Reachability				●
Navigation	●	●		●
Geo-spatial	●	◐[b]		
Text	●	●		●

[a]Comparisons should be anchored at a property
[b]Not all Geo-spatial operations are supported

Table 3: Back-end support for categories of expressions

For instance, the following query selects all products where a certain point is located within its availability region:

```
from Product p select p.name
where #point(2.0 3.0) in p.availabilityRegion
```

The **in**-operator is overloaded to also work on polygons, as is shown in the following example:

```
from Product p select p.name
where
  #polygon((2.0 2.0, 3.0 2.0, 3.0 3.0, 2.0 3.0, 2.0 2.0))
    in p.availabilityRegion
```

Another operator is `distance`, which computes the distance between two points. As an example, here is a query that selects all Review ids whose distance to a certain point is less than 200.0:

```
from Review r select r.@id
where distance(#point(2.0 3.0), r.location) < 200.0
```

All currently supported back-ends of TyphonQL are able to store the data types of Table 2, with the exception of Neo4J which does not have support for the geo-spatial polygon type, and Cassandra which has no support for geo-spatial data types at all. Table 3 furthermore shows how expression categories are supported by each back-end.

The most notable observation here is that reachability queries (involving transitive closure) is only supported on the Neo4J back-end, since on other databases it would require the (very inefficient) simulation of this feature. Cassandra is a key-value database, which, as per the mapping facilities of TyphonML, only stores primitive attributes, which obviates the need for navigation facilities (cf. Section 6.1). Furthermore it has very limited support for textual queries.

```
customdatatype address {
  street: string[256],
  city: string[256],
  zipcode: zip,
  location: point
}
```

```
customdatatype zip {
  nums: string[4],
  letters: string[2]
}
```

Figure 3: Example custom data types in TyphonML

```
entity User {
  ...
  billing: address
}
```

$\Longrightarrow$

```
entity User {
  ...
  billing$street: string[256]
  billing$city: string[256]
  billing$zipcode$nums: string[4]
  billing$zipcode$letters: string[2]
}
```

Figure 4: Inlining custom data type definitions in TyphonML entities

4.2 Custom Data Types

TyphonML supports the definition of *custom data types*: reusable abstractions to group sets of attributes. In this section we detail how TyphonQL supports this feature.

Figure 3 shows two example custom data types. On the left, the type `address` groups a set of properties to store an address. Custom data types can be nested: the `address` type contains the `zip` type in the attribute `zipcode`. The `zip` type defines the structure of (Dutch) postal codes.

TyphonQL has literal syntax for assigning attributes that have a custom data type as their type. For instance, the following insert statement inserts a `User` entity into the polystore with nested literals for both the `address` and `zip` data types:

```
insert User {
  name: "Jurgen",
  location: #point(2.0 3.0),
  billing: address(
    street: "Seventh",
    city: "Ams",
    zipcode: zip(nums: "1234", letters: "ab"),
    location: #point(2.0 3.0)
  )
}
```

Custom data types are convenient for TyphonML designers, but do not add expressive power. In other words, they can be translated away. Although some back-ends support native definition of user defined types, eliminating them by translating is preferable, because it is portable across all back-ends.

The first part of this normalization is to inline them in the TyphonML model itself. This is shown in Figure 4 for the `billing` attribute which has the type `address` of Figure 3. The rigth-hand side shows the result of the normalization. Basically, the translation inlines every custom data type, and hoists the nested attributes to the top-level, concatenating the path to it in the new attribute name. The concatenation is done using `$` (unavailable to users) to avoid collisions with existing attribute names.

Queries are normalized accordingly. For instance, when a where-clause refers to the numbers of the zip code of a user, e.g., as in `u.billing.zipcode.nums`, the TyphonQL compiler will replace this with `u.billing$zipcode$nums`. Similarly, for insert and update-statements, the nested data type literals are flattened to top-level assignments of the hoisted attributes. For instance, the insert-statement above is normalized to the following:

```
insert User {
  name: "Jurgen",
  location: #point(2.0 3.0),
  billing$street: "Seventh",
  billing$city: "Ams",
  billing$zipcode$nums: "1234"
  billing$zipcode$letters: "ab",
  billing$location: #point(2.0 3.0)
}
```

A prime advantage of these normalizations is both the mapping to back-end schemas, query partitioning and query- and update execution is unaffected.

5 Aggregation

The query partitioning of TyphonQL over various back-ends works, because filtering distributes over cartesian product (under set-based semantics). In other words, let filter be an operation that removes elements from a set based on some predicate, then the following holds:

$$\text{filter}(x \times y) = \text{filter}(x) \times \text{filter}(y)$$

That is the reason we can push where-clause evaluation to native back-ends.

Unfortunately, this is not the case for aggregation and pagination (i.e. `limit`). For instance, you can only know what to limit to after you have applied the final filtering. And for group-by, it is only possible at the very end to perform the grouping and apply the aggregation functions.

So therefore, the aggregation feature is executed in two phases. The TyphonQL query is split in two parts, a selection/filtering slice and an aggregation slice. The first part corresponds to what is partitioned over back-ends in our iteration architecture. Then, however, instead of producing the final result table in the innermost loop (see Appendix A), we process the aggregation part using the Java 8 streaming API. In other words, first we obtain the relevant records based on the select-clause, filtered by the where-clauses by recombining intermediate results from the back-ends, and *then* apply group-by partitioning and aggregation on the combined result.

For instance, here is a query to find all Dutch customers, sorted by name, and the number of products they have bought, if this number is higher than 5.

```
from Customer c, Product p
select c.name, count(c.bought)
where c.country == "NL", c.bought = p
```

```
group by c.name
having count(c.bought) > 5
order by c.name
```

This query can be sliced into the filtering part (before grouping):

```
from Customer c, Product p
select c.name, c.bought
where c.country == "NL", c.bought = p
```

On the other hand, the aggregation slice for this query is run on the (synthesized) result of the filtering slice above. In other words it is as if it runs on an inferred, ephemeral TyphonML entity of the following form, derived from the `select`-clause `c.name, c.bought`. This could be modeled in TyphonML as follows:

```
entity Result {
  c.name: text
  c.bought -> Product
}
```

The aggregation slice can then be formulated as follows:

```
from Result r
select r.name, count(r.bought)
group by r.name
having count(r.bought) > 5
order by r.name
```

Because the result of a TyphonQL query is always in a flat, tabular form, the result set of a TyphonQL query can be seen as a stream of records/tuples. The aggregation slice is then executed using the standard functions of the Java 8 streaming API on top of that stream.

Using the Java 8 streaming API for performing aggregation has a number of advantages:

- Performance: Java 8 streams are highly optimized and can potentially be parallelized.
- Modularity: the current compilation pipeline is mostly unaffected, because of the query slicing described above.
- Simpliciy: it avoids interaction effects of special casing certain situations where parts of the aggregation features could be executed natively.

Local Aggregation The above solution for aggregation in TyphonQL relatively simple to implement. It does however come at an performance cost of performing aggregation in Java. Therefore, in the case of strictly local aggregation – that is, when a TyphonQL query (including aggregation features), would land on a single back-end – we avoid this additional step, and use the native aggregation capabilities of a back-end directly, if any (at least in the case of MariaDB and MongoDB).

```
entity User {                 entity User {
  photo: blob                   user__Media :->
  avatar: blob                     User__Media[1]
}                             }

keyvaluedb Media {            entity User__Media {
 elements {              ⇒      photo: blob
  User {                        avatar: blob
    userKey -> (              }
     "User.photo",
     "User.avatar" ) }        keyvaluedb Media {
 }                              entity User__Media
}                             }
```

Figure 5: Expanding key-value store mappings to dependent entities

6 Additional Back-Ends

6.1 Key-Value Stores

Key-value stores typically support storing sets of primitively typed properties, identified by a key to allow efficient retrieval. In this section we describe how key-value stores are supported by TyphonQL. The specific back-end used for this is Cassandra[2], a widely supported key-value store, with an expressive query language CQL.

Key-value stores are special in Typhon, however, since TyphonML does not allow arbitrary entities to be mapped to a key-value store (as in the case of document stores and relational stores). Instead, the TyphonML designer can indicate that specific subsets of properties of existing entities should be stored in a key-value store.

The left-hand side of Figure 5 shows a simple TyphonML model that features a User entity with two attributes (`photo` and `avatar`), both with the type **`blob`**. In the mapping part below, both attributes are mapped to the key-value store `Media`, identified by the key `userKey`.

Although the limitation to map only sets of attributes to key-value stores is motivated by the fact that key-value stores typically allow only simple primitive types of data, it is nonetheless natural to see the mapping as the mapping of an anonymous, *inferred* entity.

This is shown on the right-hand side of Figure 5. In this case, the specific subset of attributes (`photo` and `avatar`) is lifted out of the the `User` entity, and put in a special entity `User__Media`. The mapping part is adapted accordingly: the `User__Media` entity is mapped to the `Media` key-value store.

The link with the `User` entity is maintained by creating a mandatory containment relation (`user__Media`) between `User` and `User__Media`. The cardinality of `[1]` ensures that the attributes are always present, and since the relation is containment, key-value store entries will be deleted whenever a `User` entity is deleted.

The TyphonQL compiler performs the transformation of Figure 5 on its internal representation of the TyphonML model, and then rewrites queries accordingly, so that the partitioning algorithm retrieves and filters attributes on the key-value store. For the data manipulation statements, the compiler detects updating/deleting/inserting of key-value stored attributes, and produces the required key-value store modifications.

[2]`https://cassandra.apache.org/`

```
entity Concordance {
  weight: int

  source -> Product[1]
  target -> Product[1]
}
```

```
graphdb Conc {
  edges {
   edge Concordance {
    from "Concordance.source"
    to "Concordance.target"
   }
  }
}
```

Figure 6: Mapping entities to edges in graph databases in TyphonML

6.2 Graph Databases

Just like with key-value data stores (as described above), TyphonML's graph database mapping does not support arbitrary entity mapping. In this case, the key mapping information to be produced by the TyphonML designer is to indicate certain entities as *edges*. Which entities act as nodes is then derived from how such an edge is connected to its source and target.

Figure 6 shows an example of a `Concordance` entity. It has a `weight` attribute, and two mandatory cross-reference relations, `source` and `target` to `Product`. The `Concordance` entity is going to be interpreted as an edge between two products, and the `weight` attributes is going to be an edge property. The right-hand side of Figure 6 shows how this is specified: a graph database contains edges. An edge is captured as an entity, in this case `Concordance`. Each edge needs to specify its end-points using the **`from`** and `to` clauses. In this case, the edge is defined "from" `Concordance.source` and "to" `Concordance.target`. As a result, `Product` will be mapped to the graph database *node* as well. The `weight` attribute is automatically mapped to an edge attribute because entity `Concordance` will be represented as an edge.

Note that both `source` and `target` are cross-references, and not containment: deleting an edge (`Concordance`) will not cause deletion of the nodes it connects (`Product`). The TyphonQL compiler, however, does ensure that the reverse is maintained: deleting a node removes all edges it participates in.

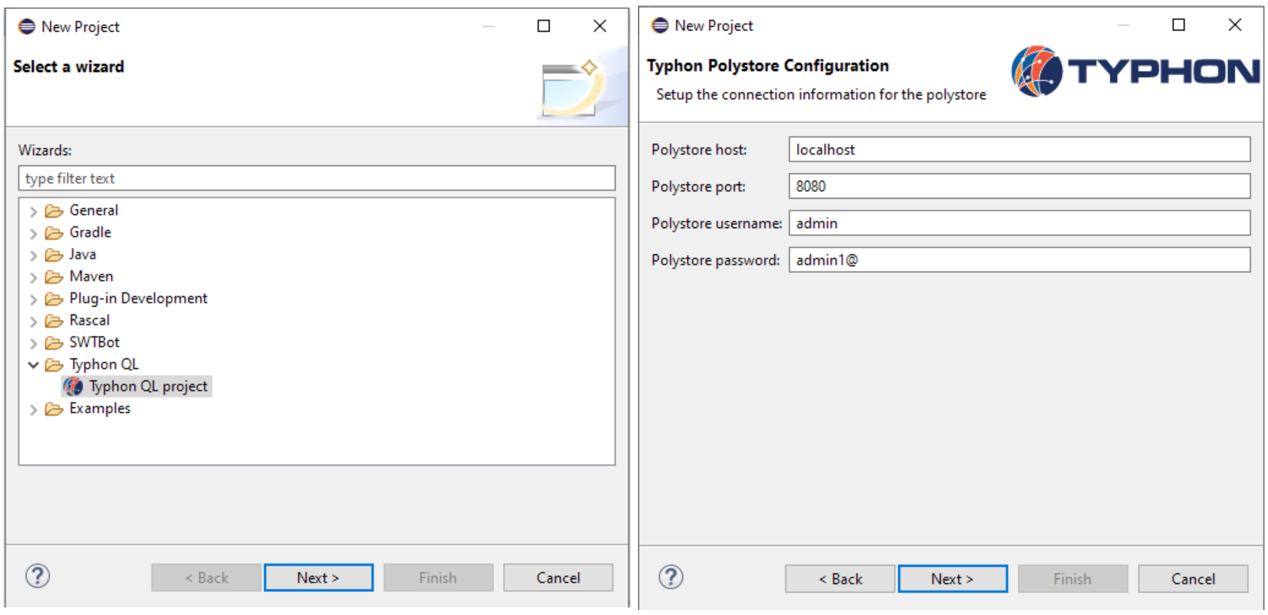
The TyphonQL compiler uses the mapping information to declare the relevant edge types in Neo4J[3], so that the identities of the product entities are used as the node identities. Attributes of edge entities become edge properties. Partitioning ensures that where-clauses related to edge entities are indeed executed on Neo4J, including advanced reachability and path queries supported by Neo4J's query language, Cypher[4].

7 TyphonQL Tools

The TyphonQL language and compiler are the central artifacts of Work Package 4. However, they are not isolated programs, but surrounded by a number of tools and frameworks to satisfy additional use case requirements and make development of TyphonQL queries easier for developers. Below we discuss the TyphonQL type checker, Eclipse IDE, and the TyphonQL AST framework for processing and analyzing TyphonQL queries in Java.

[3]https://neo4j.com/

[4]https://neo4j.com/developer/cypher-query-language/



Polystore host: localhost
Polystore port: 8080
Polystore username: admin
Polystore password: admin1@

Figure 7: Creating a new TyphonQL project

7.1 TyphonQL IDE

The TyphonQL IDE is an Eclipse Plugin that provides a TyphonQL "scratch" editor, where users can enter TyphonQL queries and statements, and observe the results. It is primarily aimed at exploring and experimenting with TyphonQL; in production, queries will be processed via the REST-based API or through the generated Data Access Layer (see D4.6 [12]). The TyphonQL IDE is built using the Language Workbench features [4] offered by the Rascal metaprogramming language [6].

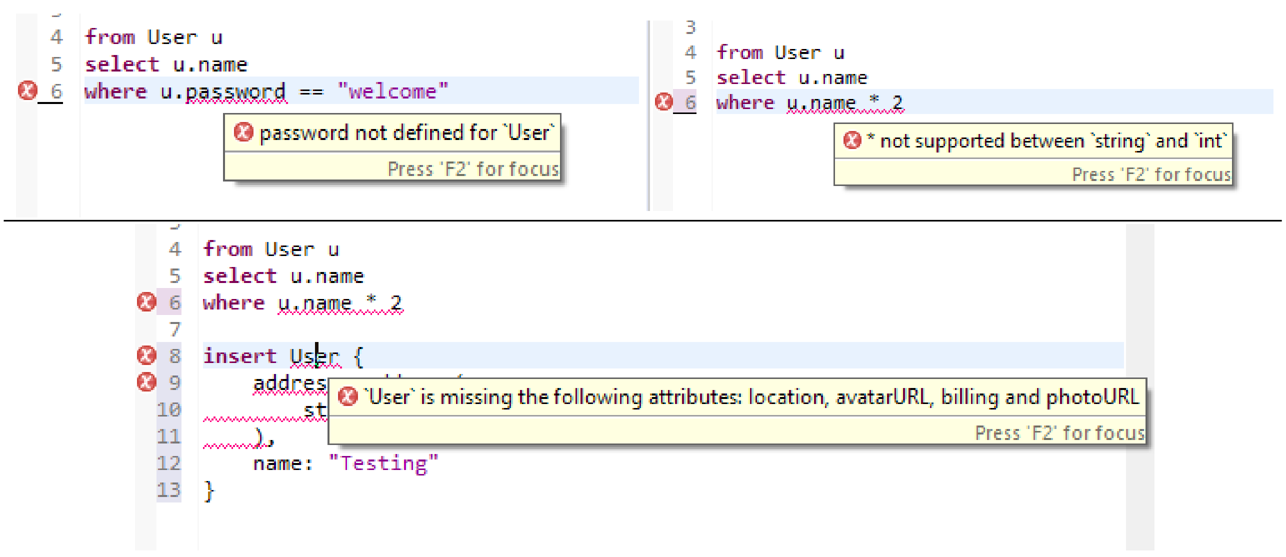
Figure 7 shows how the user – after installing TyphonQL in Eclipse – can start a new TyphonQL project. The second page of the wizard allows configuring the project with the host, port, user name, and password of a running Typhon Polystore. After completing the wizard, creating a `.tql` file inside the project will open an editor with the expected editor services such as syntax highlighting, type checkin and jump-to-definition. The queries and statements executed from the editor will be interpreted in the context of the TyphonML model that is obtained from the polystore.

Right-clicking on a query or statement will show a popup-menu with options to execute the selected snippet. In the case of a query, the result will be shown in another editor. For DML statements, a message will be shown if the statement was executed successfully. Type errors are marked within the editor by the TyphonQL type checker, which will be described next.

7.2 TyphonQL Type Checker

The type checker for TyphonQL ensures that queries entered in the TyphonQL IDE are valid with respect to the entities, attributes, data types, and relations declared in the TyphonML model. It has been implemented using Rascal's TypePal framework[5], a library for constraint-based name analysis and type checking. For performance reasons, the type checker is only run in the TyphonQL IDE. Anyway, since the IDE is used for prototyping

[5] `https://github.com/usethesource/typepal`



```
from User u
select u.name
where u.password == "welcome"
```

password not defined for `User`

Press 'F2' for focus

```
from User u
select u.name
where u.name * 2
```

* not supported between `string` and `int`

Press 'F2' for focus

```
from User u
select u.name
where u.name * 2

insert User {
    addres
        st
    ),
    name: "Testing"
}
```

`User` is missing the following attributes: location, avatarURL, billing and photoURL

Press 'F2' for focus

Figure 8: Three TyphonQL type errors marked in the TyphonQL editor

queries in the first place, this is where type checking has the most value. Furthermore, the type checker is run in our test suite, to ensure that only valid queries are tested.

The type checker performs the following checks:

- Checks that all entities listed in the **`from`**-clause are defined in the TyphonML and that no variable binding is used more than once.
- Attribute access on bindings is checked against attribute declaration in the TyphonML model.
- The types of operands to operators (e.g., $+$, $-$, **`like`**, etc) is compatible with the signature of the operator.
- Variables used in expressions are declared properly in the **`from`**-clause of a query.
- In insert- and update-statements it is checked that all attributes are specified, and all mandatory relations are assigned proper values.
- Checks that custom data type literals are used correctly.

The type checker is integrated with the TyphonQL IDE so that when a developer makes a mistake, the errors are marked inside the editor. Figure 8 shows a screenshot where the query programmer has made a number of type errors. The top-left image shows the incorrect access of a `password` attribute of the `User` entity. On the top-right side an incorrect attempt is made to multiply a string with an integer value. Finally the bottom image shows an error regarding missing attributes during insertion of a new `User`. Additionally, the type checker's name analysis is used to provide jump-to-definition hyperlinking of identifiers.

7.3 TyphonQL AST Framework

The TyphonQL language and compiler are developed in Rascal [6], a meta programming language explicitly designed for source code analysis and transformation. However, external parties might want to analyze TyphonQL queries, without relying on the Rascal language. For instance, the Analytics component of Typhon analyzes the structure of TyphonQL queries to provide detailed analytics and monitoring facilities [10].

The TyphonQL AST framework is designed to cater for this need. It consists of a code generator that generates a Java class hierarchy from the TyphonQL grammar (see Appendix B) corresponding to the abstract syntax of

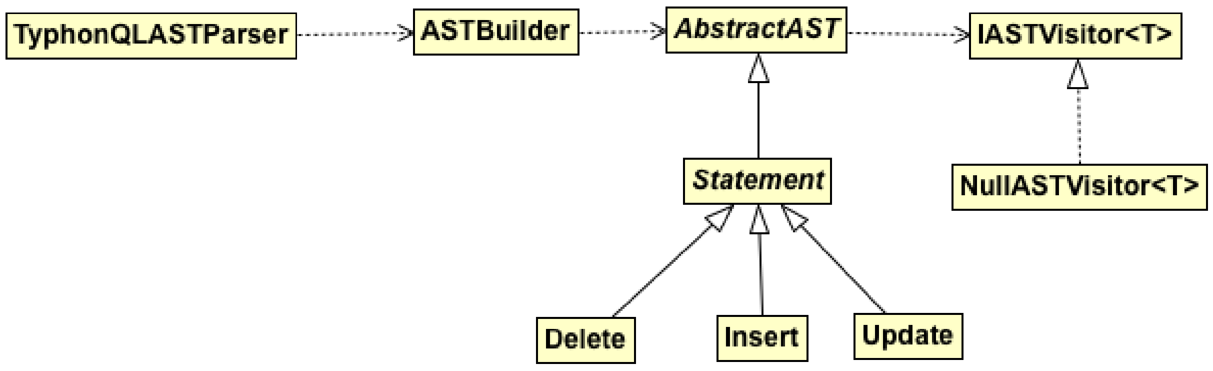


Figure 9: UML diagram of the TyphonQL AST framework (excerpt)

TyphonQL. Additionally, it provides entry points to invoke the TyphonQL parser and default Visitors [5] to traverse the resulting abstract syntax trees (ASTs).

An excerpt of the generated Java code is shown in Figure 9 in the form of a UML class diagram. The entry point is the `TyphonQLASTParser` which depends on an `ASTBuilder` class. The latter is used to construct the AST by instantiating concrete subclasses of the abstract class `AbstractAST`. Examples of such concrete subclasses are `Delete`, `Insert` and `Update` which correspond to the nonterminals of the DML sub-language of TyphonQL. Each production of the grammar included in Appendix B is mapped to a corresponding concrete AST class. To traverse a TyphonQL AST developers can implement the generic interface `IASTVisitor`, or extend `NullASTVisitor`, which provides a default traversal order.

8 Evaluation

8.1 Use Case Partner Requirements

Table 4 shows the list of use case partner requirements, relevant to Work Package 4, and their implementation status. Requirements 47 and 48 are covered by the primitive data types, as described in Section 4. Requirement 49, 50, and 62 are *sine qua non* for TyphonQL: queries can combine data from different back-ends in a single query. Requirements 51 and 58 are covered because a TyphonQL query that hits only an SQL back-end will result in plain SQL being executed. Requirements 54 and 55 are realized by the TyphonQL type checker (Section 7.2).

Requirements 53, 59, and 60 have not been implemented, because after consultation with the use case partners, support for array databases and hive structures was deemed to be of lower priority. Note, however, that the architecture of the TyphonQL compiler is structured in such a way that adding new back-ends non-invasive (because of the partitioning algorithm described in D4.4 [11]). Code completion (57) has not been implemented, but is easy to realize given the current type checker of TyphonQL. The low priority requirements 63, 66, and 67 have not been realized; text querying is currently limited to the text search features of MariaDB and MongoDB.

ID	Requirement	Priority	Status
47	The polystore query language should expose the same semantic data types and operations of the underlying database technologies as defined in their schemas or metamodels	Should	✔
48	The polystore query language should expose a relevant subset of the data types and operations from at least one of the following: MySQL version 5.5+, PostgreSQL version 7+, Java H2 version 1.4	Should	✔
49	The polystore query language shall allow querying and updating the different defined data islands	Shall	✔
50	The polystore query language shall allow interfaces between the different defined data islands	Shall	✔
51	The query language shall be able to interpret and execute SQL queries	Shall	✔
52	The query language shall be able to interpret and execute text search queries	Shall	✔[a]
53	The query language shall be able to interpret and execute array database queries.	Shall	–
54	The query language tools shall be able to check if queries are valid	Shall	✔
55	The query language tools shall be able to check if the queries are valid and compliant with the schema	Shall	✔
56	The query language tools shall support query optimisation	Shall	✔[b]
57	The query language tools should provide Intelligent code completion (auto-completion) capabilities	Should	✗
58	The system shall be able to process queries on relational databases	Shall	✔
59	The system shall be able to process queries on array databases	Shall	–
60	The system shall be able to process queries on hive structures	Shall	–
61	The system shall be able to process queries on text stores	Shall	✔
62	The system shall be able to process cross-datastore queries over all datastores of the entire system.	Shall	✔
63	The polystore query language should expose a relevant subset of the data types and operations of at least one of the following: Solr, Lucene	Should	✗
64	In the text data queries it shall be possible to search for one or more different keywords in one query	Shall	✔
65	Text data queries using patterns or full text search shall be supported	Shall	✔[c]
66	The text data queries may be able to autocorrect words	May	✗
67	The text data queries may be able to recognize incorrect spellings and mark them	May	✗

[a]Text search queries are supported through standard "like"-based operators on MariaDB and MongoDB.

[b]Optimisations performed by TyphonQL include: ordering partitioning based on filter weight (Section 3.1), special-casing `@id`-based update and delete statements; native casdcade delete handling by MariaDB and MongoDB; index generation on reference and geo-spatial fields in MongoDB and all junction tables in MariaDB; and native aggregation if it's strictly local to a back-end.

[c]This is captured by the "like"-based operators mentioned at requirement 52.

Table 4: Use Case Partner Requirements regarding WP4

ID	Requirement	Priority	Status
29	TyphonQL shall support relational SQL-like querying.	Shall	✔
30	TyphonQL shall support querying of document and key-value stores.	Shall	✔
31	TyphonQL shall support navigation-based queries such as path, reachability, and transitive closure expressions.	Shall	✔[a]
32	TyphonQL queries shall be type checked against TyphonML.	Shall	✔
33	The TyphonQL engine shall support normalization of natural language fragments to enable "querying modulo spelling".	Shall	✔
34	The TyphonQL engine shall synchronize update/delete of replicated data across backends.	Shall	✔
35	The TyphonQL engine shall be accessible through a JDBC-like API in Java.	Shall	✔
36	TyphonQL shall support querying textual data.	Shall	✔
37	The TyphonQL engine shall generate monitoring events on read and write access of tables/stores/models etc. for analytics purposes.	Shall	✔
38	TyphonQL should inform the user if a query is impossible or inefficient in their current form because of constraints on the partitioning over backends.	Should	✔
39	TyphonQL should be able to explain to the user the query plan generated for a query.	Should	✔[b]
40	The TyphonQL engine may be able to tweak the quality of partitioning based on the combination of backend and redundancy annotations in TyphonML.	May	✗
41	TyphonQL may allow transactions when possible relative to the TyphonML backend annotations.	May	✗
53	The TyphonQL execution engine shall publish data access and update requests and events to the analytics and monitoring framework's distributed messaging channel	Shall	✔
55	The TyphonQL execution engine may support rejection of data access and update requests based on feedback from analytics facilities	May	✗
65	TyphonQL shall support DDL-like (e.g., create table, alter table) operators	Shall	✔

[a]Currently, TyphonQL supports fixed-length path-based navigation expressions and reachability expression in graph-databases.

[b]The script of Figure 1 is an example of such a query plan.

Table 5: Technology Requirements TyphonQL (WP 4)

8.2 Technology Requirements

This section discusses coverage of the TyphonQL technology requirements. The requirements are summarized in Table 5. Below we describe in more detail how each requirement was addressed.

Requirements 29 and 30 are realized by the MariaDB and MongoDB back-ends (discussed in D4.4 [11]) and key-value store support as described in Section 6.1 of this deliverable. Requirement 31 is realized by allowing arbitrary navigation across entities using dot-notation, as well as reachability queries over edges in graph databases. Requirement 32 is addressed by the type checker of TyphonQL (Section 7.2). TyphonQL stores both ends of a bidirectional relation between entities; this is a kind of replication to support more efficient querying , and is an instance of requirement 34; similarly entities that play the role of edge endpoints (see Section 6.2) are partially replicated in the graph database. However, full replication of data is not supported. Requirement 33 has been addressed by D5.4 [13] and D5.6 [14], since the NLP processing engine lemmatizes freetext contents.

The JDBC-like API for TyphonQL is subsumed by the REST API, which allows sending query strings to the TyphonQL engine and retrieving the results. Textual data – requirement 36 – is supported through "like"-based operators on text fields in MariaDB and MongoDB. Requirements 37 and 53 are addressed as the Polystore API intercepts queries and sends them to the analytics framework, which analyzes them using the TyphonQL AST framework discussed in Section 7.3. Requirement 38 is covered by the type checker which will indicate if certain constructs are infeasible to realize given the mapping of entities to databases; for instance, using a reachability expression over entities that are not stored in a graph database, will be flagged as problematic. Similarly, the user is informed if certain geo-spatial expressions that are not supported by the back-end are being used. Figure 1 is an example of a "Typhon Query Plan", so this satisfies requirement 39. Finally, requirement 65 is satisfied by TyphonQL's support for evolution operators, as described in D4.4 [11].

Low priority (MAY) requirements 40, 41 and 55 have not been realized.

9 Conclusion

The Typhon project aims to offer a comprehensive infrastructure for the design, deployment, management, querying and evolution of hybrid polystores for big data. The TyphonQL compiler is a key component of this infrastructure, as it is the locus where the high-level design models of TyphonML and high-level query language TyphonQL interact with native back-ends, such as MariaDB, Cassandra, MongoDB, and Neo4J. This deliverable complements earlier deliverables D4.2, D4.3, and D4.4 to describe the final version of the TyphonQL language and compiler. In particular, it provides additional detail on specific challenges and achievements (Section 3, summarizes the final set of primitive data types and custom data type support (Section 4), the aggregation architecture (Section 5), two additional back-ends (Section 6), and the tooling ecosystem around TyphonQL (Section 7). The deliverable concludes with evaluating the current state of TyphonQL against the requirements of the use case partners and the technology requirements (Section 8).

References

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

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

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

[4] Sebastian Erdweg, Tijs van der Storm, Markus Volter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriel Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24 – 47, 2015.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented sofware*. Addison Wesley, 2009.

[6] P. Klint, T. van der Storm, and J.J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *SCAM'09*, pages 168–177. IEEE Computer Society, 2009.

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

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

[9] The University of Namur. D6.3 – Hybrid Polystore Data Migration Tools, 2018.

[10] University of York. D5.5 – Event Publishing and Monitoring Architecture (Final Version), 2019.

[11] Centrum Wiskunde & Informatica (CWI) & SWAT.engineering. D4.4 – TyphonQL Compilers and Interpeters (Interim Version), 2019.

[12] Centrum Wiskunde & Informatica (CWI) & SWAT.engineering. D4.6 – Data Access Layer Generator, 2020.

[13] Edge Hill University. D5.4 – Text Processing Pipelines (Interim Version), 2019.

[14] Edge Hill University. D5.6 – Text Processing Pipelines (Final Version), 2020.

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

A Iteration Architecture

The way intermediate results from native back-ends of partitioned queries are processed and recombined lies at the core of how the TyphonQL compiler executes queries. This appendix details the iteration architecture that underlies TyphonQL query execution.

A.1 Example

Consider and example where we have two entities, `User` and `Review`, where the first is mapped to SQL and the second to MongoDB. Users own reviews through a containment reference `reviews`. Let's assume the table for users is as follows:

@id	name	age
#p	Pablo	30
#d	Davy	20

We have the following junction table modeling the containment relation `reviews` from User to Review:

R.a	U.r
#r1	#p
#r2	#d
#r3	#d

Reviews are stored in the MongoDB collection "Review":

```
{_id: #r1, author: #p, text: "", stars: 3}
{_id: #r2, author: #d, text: "Bad", stars: 0}
{_id: #r3, author: #d, text: "Good", stars: 5}
```

Now consider we have the following query:

```
from User u, Review r select u.name, r.stars where u.reviews == r, r.text == ""
```

The intermediate result obtained from the SQL database (including the identities `@id` by default):

@id	name	U.r
#p	Pablo	#r1
#d	Davy	#r2
#d	Davy	#r3

The intermediate result from MongoDB should include all reviews with empty text where `_id` matches up with `U.r` in the SQL result:

```
{_id: #r1, stars: 3}
```

And the final result should be:

name	stars
Pablo	3

So this means that, for every row of the SQL result that we execute the MongoDB query for (and that does not return an empty set), we need to add a result to the final result. And this is the reason the iteration architecture involves nested loops of back-end query invocations. The "join" on the review's id (`U.r` and `_id`) happens within the for-loops of the iteration architecture. This avoids looping again over the individual result sets to do the join programmatically in Java.

A.2 Direct-Style Iteration

For every database back-end, we have an iterator object, which returns (partial) records/tuples, keyed by `Entity.Var.field` labels, which might end-up in the final result, and are used to do interpolation into queries using `${}`; cf. Figure 1.

The session architecture basically interprets the script, which contains "steps" executed on particular back-ends. So we may assume we have MongoDBEngine and MariaDBEngine etc., which have methods to execute a query that returns an iterable object producing record/tuple values.

Without loss of generality, we may assume each step is of the following form: `step(Backend, QueryString, map[str, Label])`. For now, we may further assume, that there's only a single step *per back-end* (not per back-end *type*, but per back-end). The script is ordered. Let's say we have steps S_0 to S_n.

We would then need the following flow of control: execute each step on its back-end, iterate over the resulting records and propagate them to the next round, effectively creating a cartesian product between back-end results:

```
for (Record $r_0$: Engine_$S_0$.exec(interpolate($S_0$.query, $S_0$.params, []))) {
  for (Record $r_1$: Engine_$S_1$.exec(interpolate($S_1$.query, $S_1$.params, [$r_0$])) {
     ...
        for (Record $r_n$: Engine_$S_n$.exec(interpolate($S_{n-1}$.query, $S_{n-1}$.params, [$r_0,...,r_{n-1}$]))) {
           appendResultRow(project([$r_0,...,r_n$], colSpec))
        }
     ...
   }
}
```

A record in this setting is a mapping from `Entity.Var.Field` keys to (scalar) values. One could say it is a subset of fields of a row in the final result of the query. In each loop we horizontally concatenate the records until `appendResultRow` adds the complete row to the final result, according to the expressions in the select-clause of the query.

The interpolate function substitutes parameters the query string, taking values, from the list of currently produced records (i.e. the current row).

In the inner most loop, `appendResultRow` should project out the relevant `Entity.Var.Field` components from r_0 to r_n that are required for constructing the final result table, the signature of which derives from the select-clause of the original TyphonQL query.

A.3 Inversion of Control

Currently, the script is interpreted on top of a session abstraction from within Rascal. This means that the above structure needs to be inverted, in the sense that the for-loops cannot be "driving" the interpretation (pull-style), but have to incrementally run through inversion of control (push-style).

The below pseudo-Java code models a Session object that constructs the same control-flow as the code above, but lazily, by using an API method that is called from within Rascal for every step in the script. It essentially constructs a nested structure of closures which are only "forced" when the final result is requested.

```
class Session {
  private List<Consumer<List<Record>>> script = [];

  public void engine_S0_exec(String q, Params params) {
    final int nxt = script.size() + 1;
    script.add((List<Record> row) -> {
       for (Record myR: Engine_S0.exec(interpolate(q, params, row))) {
          script.get(nxt).call(row ++ myRecord);
       }
    });
  }

  public void engine_S1_exec(...) { /* similar */ }

  ...

  List<List<Record>> read(ColSpec columns) {
    List<List<Record>> result = [];
    script.add((List<Record> row) -> {
       result.append(project(row, columns));
    });
    script.get(0).call([]);
    return result;
  }
}
```

The `script` field contains a sequence of closures that each invoke the next one passing in results that have been generated, until the last closure is reached which constructs the result table.

The `read` method should be called last with a specification of the actual `Entity.Var.Field` labels that are derived from the select clause of the original query.

B Full Grammar of TyphonQL

B.1 Queries

```
module lang::typhonql::Query

extend lang::typhonql::Expr;
```

```
syntax Query
  = from: "from" {Binding ","}+ bindings "select" {Result ","}+ selected
       Where? where GroupBy? groupBy OrderBy? orderBy;

syntax Result
  = aliassed: Expr!obj!lst expr "as" Id attr
  | normal: Expr expr // only entity path is allowed, but we don't check
  ;

syntax Binding = variableBinding: EId entity VId var;

syntax Where = whereClause: "where" {Expr ","}+ clauses;

syntax GroupBy = groupClause: "group" {Expr ","}+ exprs Having? having;

syntax Having = havingClause: "having" {Expr ","}+ clauses;

syntax OrderBy = orderClause: "order" {Expr ","}+ exprs;
```

B.2 Data Manipulation

```
module lang::typhonql::DML

extend lang::typhonql::Expr;
extend lang::typhonql::Query;


syntax Statement
  = \insert: "insert" {Obj ","}* objs
  | delete: "delete" Binding binding Where? where
  | update: "update" Binding binding Where? where "set"  "{" {KeyVal ","}* keyVals "}"
  ;

// extension for update: not to be used in insert
syntax KeyVal
  = add: Id key "+:" Expr value
  | remove: Id key "-:" Expr value
  ;
```

B.3 Expressions

```
module lang::typhonql::Expr

extend lang::std::Layout;
extend lang::std::Id;

syntax Expr
  = attr: VId var "." {Id "."}+  attrs
  | var: VId var
  | placeHolder: PlaceHolder
  | key: VId var "." "@id"
  | @category="Number" \int: Int intValue
  | @category="Constant" \str: Str strValue
  | @category="Number" \real: Real realValue
  | @category="Constant" \dt: DateTime dtValue
  | @category="Constant" point: Point pointValue
  | @category="Constant" polygon: Polygon polygonValue
  | \bool: Bool boolValue
  | uuid: UUID uuidValue
  | blob: BlobPointer blobPointerValue
  | bracket "(" Expr arg ")"
  | obj: Obj objValue // for use in insert and allow nesting of objects
  | custom: Custom customValue // for use in insert and allow nesting of custom data types
  | refLst: "[" {UUID ","}* refs "]" // plus to not make amb with empy lst
  | null: "null"
  | pos: "+" Expr arg
  | neg: "-" Expr arg
  | call: VId name "(" {Expr ","}* args ")"
  | not: "!" Expr arg
  > left (
      left mul: Expr lhs "*" Expr rhs
    | left div: Expr lhs "/" Expr rhs
  )
  > left (
      left add: Expr lhs "+" Expr rhs
    | left sub: Expr lhs "-" Expr rhs
  )
  > non-assoc (
      non-assoc hashjoin: Expr lhs "#join" Expr rhs
    | non-assoc eq: Expr lhs "==" Expr rhs
    | non-assoc neq: Expr lhs "!=" Expr rhs
    | non-assoc geq: Expr lhs "\>=" Expr rhs
    | non-assoc leq: Expr lhs "\<=" Expr rhs
    | non-assoc lt: Expr lhs "\<" Expr rhs
    | non-assoc gt: Expr lhs "\>" Expr rhs
    | non-assoc \in: Expr lhs "in" Expr rhs
    | non-assoc like: Expr lhs "like" Expr rhs
  )
  > left intersect: Expr lhs "&" Expr rhs
  > left and: Expr lhs "&&" Expr rhs
  > left or: Expr lhs "||" Expr rhs
  ;


// Entity Ids
lexical EId = Id entityName \ Primitives;

keyword Primitives
  = "int" | "bigint" | "string" | "bool" | "text" | "float"
  | "blob" | "freetext" | "date" | "datetime" | "point" | "polygon" ;


syntax Point
  = singlePoint: "#point" "(" XY ")"
  ;
```

```
syntax XY
  = coordinate: Real Real;

syntax Polygon
  = shape: "#polygon" "(" {Segment ","}* segments ")"
  ;

syntax Segment
  = line: "(" {XY ","}* points ")";


// Variable Ids
lexical VId = Id variableName \ "true" \ "false" \ "null";

// extend Id for customdata type inlined representation
lexical Id
  = Id "$" {Id "$"}+
  ;

lexical Bool = "true" | "false";

syntax Obj = literal: Label? labelOpt EId entity "{" {KeyVal ","}* keyVals "}";

syntax Custom = literal: EId typ "(" {KeyVal ","}* keyVals ")";

lexical Label = "@" VId label;

syntax KeyVal
  = keyValue: Id key ":" Expr value
  | storedKey: "@id" ":" Expr value
  ;

lexical PlaceHolder = "??" Id name;

// textual encoding of references
lexical UUID = @category="Identifier" "#" UUIDPart part;
lexical UUIDPart = [\-a-zA-Z0-9]+ !>> [\-a-zA-Z0-9];
lexical BlobPointer = @category="Identifier" "#blob:" UUIDPart part;

lexical Int
  = [1-9][0-9]* !>> [0-9]
  | [0]
  ;

lexical Real
  = Int "." [0]* !>> "0" Int?
  | Int "." [0]* !>> "0" Int? [eE] [\-]? Int;

syntax DateTime
  = date: JustDate date
  | time: JustTime  time
  | full: DateAndTime dateTime ;
```

```
lexical JustDate
  = "$" DatePart "$";

lexical DatePart
  = [0-9] [0-9] [0-9] [0-9] "-" [0-1] [0-9] "-" [0-3] [0-9]
  | [0-9] [0-9] [0-9] [0-9] [0-1] [0-9] [0-3] [0-9] ;


lexical JustTime
  = "$T" TimePartNoTZ !>> [+\-] "$"
  | "$T" TimePartNoTZ TimeZonePart "$"
  ;

lexical DateAndTime
  = "$" DatePart "T" TimePartNoTZ !>> [+\-] "$"
  | "$" DatePart "T" TimePartNoTZ TimeZonePart "$";

lexical TimeZonePart
  = [+ \-] [0-1] [0-9] ":" [0-5] [0-9]
  | "Z"
  | [+ \-] [0-1] [0-9]
  | [+ \-] [0-1] [0-9] [0-5] [0-9]
  ;

lexical TimePartNoTZ
  = [0-2] [0-9] [0-5] [0-9] [0-5] [0-9] ([, .] [0-9] ([0-9] [0-9]?)?)?
  | [0-2] [0-9] ":" [0-5] [0-9] ":" [0-5] [0-9] ([, .] [0-9] ([0-9] [0-9]?)?)?
  ;
lexical Str = [\"] ![\"]* [\"];
```

B.4 Data Definition

```
module lang::typhonql::DDL

extend lang::typhonql::Expr;

syntax Statement
  = \createEntity: "create" EId eId "at" Id db
  | \createAttribute: "create" EId eId "." Id name ":" Type typ
  | \createRelation: "create" EId eId "." Id relation Inverse? inverse Arrow EId target
      "[" CardinalityEnd lower ".." CardinalityEnd upper "]"
  | \dropEntity: "drop" EId eId
  | \dropAttribute: "drop" "attribute" EId eId "." Id name
  | \dropRelation: "drop" "relation" EId eId "." Id name
  | \renameEntity: "rename" EId eId "to" EId newEntityName
  | \renameAttribute: "rename" "attribute" EId eId "." Id name"to" Id newName
  | \renameRelation: "rename" "relation" EId eId  "." Id name "to" Id newName
  ;

syntax Inverse = inverseId: "(" Id inverse ")";

syntax Type
  = intType: "int" // the 32bit int
  | bigIntType: "bigint"  // 64bit
  | stringType: "string" "[" Nat maxSize "]"
  | textType: "text"
  | pointType: "point" // To check
  | polygonType: "polygon"
  | boolType: "bool"
  | floatType: "float" // IEEE float
  | blobType: "blob"
  | freeTextType: "freetext" "[" {Id ","}+ nlpFeatures "]"
  | dateType: "date"
  | dateTimeType: "datetime"
  ;

lexical Nat = [0-9]+ !>> [0-9];

lexical Arrow = "-\>" | ":-\>";

lexical CardinalityEnd = [0-1] | "*";
```