International Journal of Software Engineering
and Knowledge Engineering
Vol. 30, No. 6 (2020) 785–803
© The Author(s)
DOI: 10.1142/S0218194020400070

Exploiting Declarative Mapping Rules for Generating GraphQL Servers with Morph-GraphQL

David Chaves-Fraga*, Freddy Priyatna†,
Ahmad Alobaid‡ and Oscar Corcho§

Ontology Engineering Group
Universidad Politécnica de Madrid
Madrid, Spain
**dchaves@fi.upm.es*
†fpriyatna@fi.upm.es
‡aalobaid@fi.upm.es
§ocorcho@fi.upm.es

Received 19 December 2019
Revised 15 February 2020
Accepted 24 March 2020

In the last decade, REST has become the most common approach to provide web services, yet it was not originally designed to handle typical modern applications (e.g. mobile apps). GraphQL was proposed to reduce the number of queries and data exchanged in comparison with REST. Since its release in 2015, it has gained momentum as an alternative approach to REST. However, generating and maintaining GraphQL resolvers is not a simple task. First, a domain expert has to analyze a dataset, design the corresponding GraphQL schema and map the dataset to the schema. Then, a software engineer (e.g. GraphQL developer) implements the corresponding GraphQL resolvers in a specific programming language. In this paper, we present an approach to exploit the information from mappings rules (relation between target and source schema) and generate a GraphQL server. These mapping rules construct a virtual knowledge graph which is accessed by the generated GraphQL resolvers. These resolvers translate the input GraphQL queries into the queries supported by the underlying dataset. Domain experts or software developers may benefit from our approach: a domain expert does not need to involve software developers to implement the resolvers, and software developers can generate the initial version of the resolvers to be implemented. We implemented our approach in the Morph-GraphQL framework and evaluated it using the LinGBM benchmark.

Keywords: OBDA; Declarative Mappings; GraphQL.

1. Introduction

Introduced in 2000, Representational State Transfer (REST) has become the most common apprach to provide web services in the last decade. Those web services that conform to the REST principles, known as RESTful web services, use HTTP/S and its operations to make requests to the underlying server, such as GET to retrieve objects, POST to add objects, PUT to modify objects and DELETE to remove objects, among others.

Over the years, the complexity of modern software concept has evolved since the inception of REST. For example, typical mobile applications have to take into account aspects that receive little attention in traditional applications, such as the size of data being exchanged/transmitted and the number of API calls being made. These aspects are relevant to the problem known as *over-fetching* and *under-fetching* [1–3]. Over-fetching refers to the situation in which a REST endpoint returns more data than what is required by the developer [1–3]. For example, a developer may need some information about the name of a user, so she hits the corresponding endpoint (`/user`). However, the endpoint may return information that is not needed by the client, such as birth date and address. The opposite also raises a problem, which is having the REST endpoint provide less data than required. Such a case is called under-fetching [1–3]. It refers to the situation in which a single REST endpoint does not provide sufficient information requested by the client. For example, in order to obtain the names of all friends of a particular user, typically two endpoints may be needed: the first is the endpoint that returns the identifiers of all the friends (`/friends`), and the second is the one that returns the details of each of the friends based on the identifier (`/user`).

In order to ameliorate the aforementioned problems, Facebook proposed the GraphQL query language [4], initially being used internally by the company in 2012. GraphQL was released for public use in 2015 and since then has been adopted by companies from various sectors such as technology (GitHub), entertainment (Netflix), finance (PayPal), travel (KLM), among others.

The two main components of a GraphQL server are *schema* and *resolvers*. The GraphQL schema specifies the type of an object together with the fields that can be queried. GraphQL resolvers are data extraction functions responsible for accessing the underlying datasets. These functions are written by software engineers using a specific programming language and, are then used by GraphQL engines, which translate GraphQL queries to the corresponding underlying query language of the sources (e.g. SQL). Multiple GraphQL engines support major programming languages (e.g. JavaScript, Python, Java, Golang, Ruby). In addition to the aforementioned frameworks, query planning tools have been developed in order to translate GraphQL queries into other query languages (e.g. dataloader,[a] joinmonster[b]).

[a] `https://github.com/facebook/dataloader.`
[b] `https://join-monster.readthedocs.io/en/latest/.`

Generating a GraphQL server requires expertise from both domain experts and software developers. Typically, the following tasks need to be done:

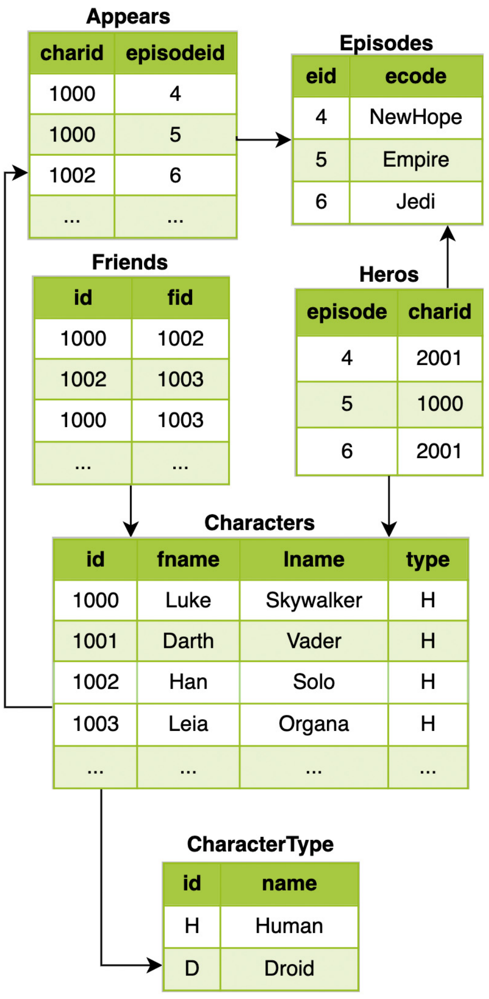
(1) A domain expert would analyze the underlying datasets, propose a unified view schema as a GraphQL schema and how the source datasets would need to be mapped into the GraphQL schema. Note that there is no standard mechanism to represent these mappings, hence, there are multiple different ways to represent them. For example, domain experts may use a spreadsheet, which is not necessarily easy to understand by another domain expert. Some of such spreadsheets may represent the relation among source and target concepts in a simple two-column-based approach. Others use Excel files with pages, such that each page represents a concept. Others add ids instead of property names. It becomes even more messy when there is an operation involved (e.g. source has the name in two properties/columns, "first-name" and "last-name" while the target has one single attribute to represent both, "name").
(2) A software developer would then implement those mappings as GraphQL resolvers, a process that takes significant resources. Given that the complexity of any given source code grows faster than the size of the source code, generating GraphQL resolvers would become more difficult even for a standard-sized dataset which typically contains more than a handful tables and hundreds of properties. This situation might worsen if the underlying dataset evolves, considering that the corresponding resolvers have to be updated as well. GraphQL resolvers may not be easily understood by other developers who were not involved in the initial version, thus bringing the possibility of introducing errors.

In this paper, we propose the exploitation of declarative mapping languages to specify the rules that relate the source datasets and the GraphQL schema. Declarative mapping languages, such as the W3C R2RML [5] and its extensions (e.g. RML [6]), have been used to generate knowledge graphs from existing datasets. The use of declarative mappings is based on the idea that a standard mapping language would facilitate a better understanding of the relationships between the underlying data source and the exposed GraphQL schema. Furthermore, they also allow for better maintainability as those mappings are independent from any programming language. Our main contribution in this paper is an approach that translates declarative mappings to GraphQL resolvers.

The rest of this paper is structured as follows: in Sec. 2, we review GraphQL and declarative mapping languages, and in Sec. 3, we describe our approach on translating declarative mappings to GraphQL resolvers. In Sec. 4, we use the recently proposed GraphQL benchmark to evaluate our implementation. Finally, related work and conclusion are presented in Secs. 5 and 6.

2. Background

In this section, we provide some background on two of the underlying concepts that we use: declarative mapping languages and GraphQL. We use the



Appears

charid	episodeid
1000	4
1000	5
1002	6
...	...

Episodes

eid	ecode
4	NewHope
5	Empire
6	Jedi

Friends

id	fid
1000	1002
1002	1003
1000	1003
...	...

Heros

episode	charid
4	2001
5	1000
6	2001

Characters

id	fname	lname	type
1000	Luke	Skywalker	H
1001	Darth	Vader	H
1002	Han	Solo	H
1003	Leia	Organa	H
...	...	...	...

CharacterType

id	name
H	Human
D	Droid

Fig. 1. Tables used in the Star Wars example, inspired by the example provided in the reference implementation.

example[c] provided in the reference implementation based on the Star Wars movies to explain the background concepts. An overview of its schema in a tabular model and some of the data are shown in Fig. 1.

2.1. *Declarative mapping languages*

Declarative Mapping Languages are well known as a way to generate knowledge graphs using the Ontology-Based Data Access (OBDA) paradigm [7]. Such

[c]`https://github.com/graphql/graphql-js/`.

mappings generate a global unified view of datasets using mappings, which specify the details of the generated knowledge graph. There are two ways in the OBDA paradigm to generate knowledge graphs from datasets: data translation and query translation [7]. In data-translation, the mapping rules are used to generate the instances of the global schema. Then, those instances are loaded in a triple store so that queries posed over the global schema can be evaluated using the corresponding query language (e.g. SPARQL). In query-translation, queries over the global schema are translated into queries over the local schema, taking into account the information provided in the mappings; in other words, the generated knowledge graph is kept virtual. In short, data-translation method transforms the data and stores it in another format (materialized) while query-translation method transforms the query while maintaining the data in their original format.

R2RML [5] is an example of declarative OBDA mappings, whose focus is the generation of knowledge graphs from relational databases. The W3C R2RML Recommendation (September 2012) allows specifying rules for transforming relational database content into an RDF-based knowledge graph. Several mapping languages have been proposed as extensions of R2RML, such as RML [6] (to deal with CSVs, JSON and XML data sources), xR2RML [8] (to deal with MongoDB), KR2RML [9] (to deal with nested data) or RMLC-Iterator (for statistical CSV files) [10].

Transformation rules are defined in these mapping languages following a similar structure. A mapping document contains a set of Triples Map (`rr:TriplesMap`[d]) that are used to generate RDF triples from the source datasets. A Triples Map consists of the following:

- A Logical Source (`rr:LogicalTable`, `rml:LogicalSource`[e]) that specifies the path or URL of the underlying source.
- A Subject Map (`rr:SubjectMap`) that specifies the rule for generating the subjects of the triples.
- A set of Predicate Object Maps (`rr:PredicateObjectMap`) that consists of a pair of Predicate Map (`rr:PredicateMap`) and Object Map (`rr:ObjectMap`) that specify rules for generating predicate and object of the triples.
- If a join with another Triples Map is needed, a Reference Object Map (`rr:RefObjectMap`) may be specified, together with the correspondence link to that Triples Map (`rr:parentTriplesMap`) and the references to perform the condition (`rr:child` and `rr:parent`).

A Term Map (`rr:TermMap`) is either a Subject Map, a Predicate Map, or an Object Map. Term Maps are used to generate RDF terms, either as IRIs (`rr:IRI`), Blank Nodes (`rr:BlankNode`), or literals (`rr:Literal`). The values of the term maps can be specified using a constant-valued map (`rr:constant`), a reference-valued map (`rr:column`, `rml:reference`), or a template-valued map (`rr:template`). Furthermore,

[d] `rr` stands for the R2RML namespace URI `http://www.w3.org/ns/r2rml#`.
[e] `rml` stands for the RML namespace URI `http://semweb.mmlab.be/ns/rml#`.

```
<TMEpisodes>
rr:logicalTable [
    rr:table 'Episodes';
];
rr:subjectMap [
    rr:template 'ex.com/episode/{eid}';
    rr:class schema:Episode
];
rr:predicateObjectMap [
  rr:predicate schema:code;
  rr:objectMap    [ rr:column  'ecode' ]
];
.
```

Listing 1. R2RML mapping for episode.

additional information such as datatype (`rr:datatype`) can also be attached to Term Maps. In Listing 1, we show the R2RML mapping for the table Episode where the subject is defined as a template involving the `eid` column and a predicate-object pair involving the `ecode` column.

2.2. *GraphQL*

GraphQL is a specification that provides a unified view for accessing heterogeneous datasets using its query language. Besides the query language, the specification defines how a GraphQL server may be implemented to allow developers to deploy their own implementation in different programming languages. In this section, we describe and provide an example of the main components of a GraphQL server: schema and resolvers.

A GraphQL schema specifies all the available types and their properties. For example, in Listing 2, we can see that the schema for the `Episode` type together with

```
type Query {
    listEpisode(identifier:String, code:String): [Episode]
    ...
}
type Episode {
    identifier:String
    code:String
}
```

Listing 2. GraphQL schema for type episode.

its two fields: `identifier` and `code`. This part of the server also specifies all the possible entry points of the instance, known as *Query root* or *Query type*.

A GraphQL resolver describes the relationship between the defined GraphQL types/fields and the data sources. It implements the methods for accessing the data of each field in a specific dataset. For example, given the dataset in Figure 1, a GraphQL resolver may provide queries for retrieving all instances of the defined GraphQL types (e.g. `listAppear, listEpisode, listCharacter, listFriends, listHeroes`).

3. The Morph-GraphQL Framework

The Morph-GraphQL framework (Fig. 2) proposes the exploitation of the information encoded in declarative mapping rules following a well-known specification (e.g. R2RML and RML) to generate GraphQL servers. A domain expert can create these

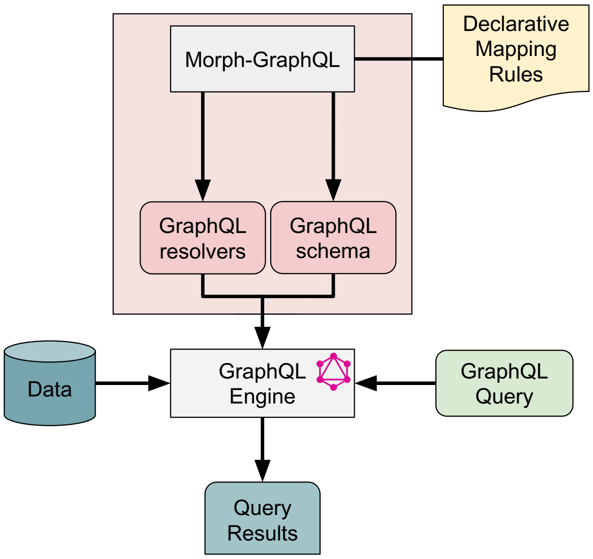


Fig. 2. **morph-GraphQL workflow.** morph-GraphQL receives declarative mappings and generates GraphQL servers. These servers, as it is defined in the specification, contain their two main components (i.e. schema and resolvers) that can be used by any GraphQL engine to evaluate queries over the data source.

types of mappings without the need for programming skills. Despite that, the creation of mappings might not be easy for domain experts to be created from scratch, there are several tools with easy to use graphical interface that already developed by researchers in the semantic Web community such as RMLEditor [11] or KARMA [12]. The generated GraphQL servers benefit from the wide range of tools available for GraphQL in order to access data stored in various formats (i.e. RDB, CSV, JSON). The approach consists of the following steps: (1) the generation of the definition of the queries to be evaluated by the underlying dataset (e.g. ListEpisodes), (2) the generation of the types in the GraphQL schema and (3) the generation of GraphQL resolvers.

Auxiliary Functions. We present here a set of auxiliary functions that will be used in the functions that generate resolvers.

- *getConstant*(*TermMap*) takes the constant `prefix:attr` in the constant-value term map where *TermMap* = `rr : constant″prefix : attr″` and retrieves its specific value *attr*.
- *getReference*(*TermMap*) retrieves the reference `ref` in the reference-value term map where *TermMap* = `rr : column“ref″` or *TermMap* = `rml : reference“ref″`.
- *getTemplate*(*TermMap*) retrieves the template `template` in the template-value term map where *TermMap* = `rr : template″template″`. In this case, the function retrieves the concatenation between the strings and the references that are part of the template, hence, the implementation of this functions depend of the underlying query system used for retrieving the data. For example, given a MySQL database and the term map `rr:template ″ex.com/episode/{eid}″` as the inputs, this function returns `CONCAT(″ex.com/episode/{eid}″, eid)`.
- *getDataType*(*ObjectMap*) that given an `rr:ObjectMap` that contains a `rr: dataType ″xsd:type″` returns the corresponding GraphQL type. For example, *getDataType*(*rr* : *dataType*“*xsd* : *string*”) returns `String`.
- *getTypeFromClass*(*SubjectMap*) that given an `rr:SubjectMap` returns the type value based on the `rr:class` property. For example, *getDataType*(*rr* : *class*“*foaf* : *Person*”) returns `Person`.

3.1. *Generating queries*

We present a set of translation functions that convert a triples map into the corresponding query to be used in GraphQL resolvers. This set of functions is adapted from the work presented in [13], originally proposed to translate SPARQL queries into SQL queries without the presence of any mappings. For example, given Listing 1 as the input, these functions generate the SQL query shown in variable *sql* in Listing 3.

- α(*TriplesMap*) returns a set of logical sources associated with the triples map *TriplesMap*, which is the logical source associated to the triples map *TriplesMap* and additionally all the referenced source if *TriplesMap* contains Referenced Object Maps.
- β(*TermMap*) that given a term map *TermMap* returns the corresponding query expression, that is:

– *getConstant*(*TermMap*) if *TermMap* is a constant-value map
– *getReference*(*TermMap*) if *TermMap* is a reference-value map
– *getTemplate*(*TermMap*) if *TermMap* is a template-value map.

```
listEpisode: function({identifier,code}) {
 let sql= 'SELECT ex.com/episode/' || eid AS c1, ecode AS c2
   FROM episodes WHERE c1 = ${identifier} AND c2 = ${code}'
 let data = db.all(sql);
 let allInstances = [];
 return data.then(rows => {
   rows.forEach((row) => {
     let instance = new Episode(row['c1'], row['c2']);
     allInstances.push(instance);
   });
   return allInstances;
 });
}
```

Listing 3. GraphQL resolver for type episode.

- *alias*(*TermMap*) generates a unique alias to be used in the query generation.
- *genPR*(*TriplesMap*) generates a query expression which projects the relevant query expressions of a triples map *TriplesMap* (i.e. β of Subject Map and all Object Maps) together with their aliases.
- *genCond*(*TriplesMap*) generates a query expression which is evaluated to true if they match the arguments passed in the resolver functions and additionally the join conditions if *TriplesMap* contains Referenced Object Maps.
- Finally, *trans*(*TM*) builds the valid query statement using the results of the previous functions. For example, in the case of an SQL database, *trans*(*TM*) = "`SELECT` *genPR*(*TM*) `FROM` α(*TM*) `WHERE` *genCond*(*TM*)" translates a *TM* into the corresponding query.

3.2. *Generating schema*

The generation of the Schema for GraphQL is divided into two steps. The first one is focused on the generation of the possible entry points of the server (i.e. the query root) and the second one generates the Types defined for the server. Exploiting the mapping rules as inputs, Morph-GraphQL automatically generates both components.

3.2.1. *Generating query root*

First, Morph-GraphQL generates the entry points of the server. Algorithm 1 shows how this step is executed. Taking as input a mapping document, it iterates over the TriplesMap and extracts the information needed for defining the entry points: the type value extracted from the class property of the subject map and the set of attributes together with the types extracted from the predicate-object maps.

Algorithm 1 GenerateQueryRoot(Mapping)

```
$queryRoot.init()$
for all $TriplesMap \leftarrow Mapping$ do
    $typeClass = getTypeFromClass(TriplesMap.getSubjectMap())$
    $poms = TriplesMap.getPredicateObjectMaps()$
    for all $pom \leftarrow poms$ do
        $datatype = getDataType(pom.getObjectMap())$
        $attribute = getConstant(pom.getPredicateMap())$
        $attributes.add(attribute, datatype)$
    end for
    $query = createListQuery(typeClass, attributes)$
    $queryRoot.add(query)$
end for
return $queryRoot$
```

Morph-GraphQL is able to generate automatically a set of basic entry points (e.g. ListEpisodes, ListCharacter) that can be filtered by each attribute.

3.2.2. *Generating types*

Algorithm 2 generates a GraphQL type from a Triples Map. It generates a GraphQL type *typeClass*, where *typeClass* is the class specified in the Subject Map of the Triples Map. The fields of the *typeClass* are all the mapped predicates in the

Algorithm 2 GenerateType(TriplesMap)

```
$type.init()$
$typeClass = getTypeFromClass(TriplesMap.getSubjectMap())$
$type.add(typeClass)$
$poms = TriplesMap.getPredicateObjectMaps()$
for all $pom \leftarrow poms$ do
    $datatype = getDataType(pom.getObjectMap())$
    $attribute = getConstant(pom.getPredicateMap())$
    $type.add(createAtttribute(attribute, datatype))$
end for
return $type$
```

Predicate Object Maps of the Triples Map. The datatypes of the fields are the results of function *getDataType*, which returns the corresponding GraphQL type from the datatype specified in the Object Maps of the Triples Map. This function is called for each Triples Map defined in the mapping document.

3.3. *Generating resolvers*

Algorithm 3 generates a GraphQL resolver from a Triples Map. Based on the entry

Algorithm 3 GenerateResvolver(TriplesMap)

```
$resolver.init()$
$typeClass = getTypeFromClass(TriplesMap.getSubjectMap())$
$poms = TriplesMap.getPredicateObjectMaps()$
for all $pom \leftarrow poms$ do
    $attribute = getConstant(pom.getPredicateMap())$
    $attributes.add(attribute)$
end for
$resolver.add(defineListQueryFunction(typeClass, attributes))$
$resolver.add(translateQuery(trans(TriplesMap))$
$resolver.add(execute(query, rdb))$
$resolver.add(constructResults(attributes, queryResults))$
return $Resolver$
```

points defined in the schema, for each Triple Map, Morph-GraphQL generates the `list`*typeClass* resolver. First, it defines the function for querying the Type *typeClass* with the attributes defined in the mapping. Then, the algorithm uses the functions defined in Sec. 3.1 (*trans* function) to translate the query to the underlying query language adapting the approach defined in [13]. These two steps use the auxiliary functions $getRerereference()$ and *getTemplate*() in order to obtain the correct references of the data source columns/keys from the mapping rules. Finally, it defines the manner how the engines have to execute the query on the underlying database engine and generate the corresponding instances by calling the constructor of Type *typeClass*.

4. Evaluation

In this section, we present the experimental evaluation of Morph-GraphQL. Our aim is to answer the following questions:

RQ1: Can Morph-GraphQL generate a GraphQL server from declarative mappings that is able to answer the set of queries provided by a GraphQL benchmark?

RQ2: Is there any significant difference between for the queries that can be answered by the generated GraphQL server in terms of response time between GraphQL queries and their equivalent SPARQL queries posed over the RDF dataset generated by the same declarative mappings?

We have implemented our framework as an open-source project **Morph-GraphQL**.[f,g] In our previous work [14], we described the full example of Star Wars generating a GraphQL server based on an R2RML mapping using Morph-GraphQL. Currently, the Morph-GraphQL framework is able to translate R2RML mappings into a Javascript-based GraphQL server for accessing tabular datasets (CSV files or Relational Databases). We use the JoinMonster library[h] to generate efficient SQL queries when joins are needed.

4.1. *Linköping graphQL benchmark*

Currently, the only GraphQL benchmark available is the Linköping GraphQL Benchmark (LinGBM), proposed by Hartig *et al.* [15]. This benchmark focuses on exposing read-only GraphQL APIs over a legacy relational database. At the time of writing, the LinGBM benchmark v1.0 sets its context in the domain of e-commerce. It consists of a dataset generator and a set of query templates (a query with place-holder variables to be instantiated). Additionally, guidelines are provided on the mapping between the relational database schema and the GraphQL schema (e.g. Table Offer is mapped to GraphQL type Offer).

Dataset Generator. The dataset generator[i] is based on the Berlin SPARQL Benchmark (BSBM) [16]. The dataset contains 10 tables (e.g. Vendor, Offer, Producer, Product, and Person, Review) with different join cardinalities (e.g. 1-1, 1-N, M-N).

Choke-points and Queries. The benchmark includes a list of *choke-points*, which are challenges that have been identified for answering GraphQL queries. This is done following the design methodology for benchmark development[j] introduced by the Linked Data Benchmark Council.[k] Five classes of choke-points that are proposed are as follows:

(1) Choke Points Related to Attribute Retrieval. (1 choke-point)
(2) Choke Points Related to Relationship Traversal. (5 choke-points)
(3) Choke Points Related to Ordering and Paging. (3 choke-points)
(4) Choke Points Related to Searching and Filtering. (5 choke-points)
(5) Choke Points Related to Aggregation. (2 choke-points)

These choke-points are covered in the 16 handcrafted query templates provided by the benchmark. The summary of the queries, the relation with the choke-points and the support of Morph-GraphQL are shown in Table 1.

[f] `https://github.com/oeg-upm/morph-graphql.`
[g] `https://doi.org/10.5281/zenodo.3584339.`
[h] `https://join-monster.readthedocs.io`
[i] `https://github.com/LiUGraphQL/LinGBM/wiki/Datasets.`
[j] `http://ldbcouncil.org/blog/choke-point-based-benchmark-design.`
[k] `http://ldbcouncil.org/.`

Table 1. Supported queries.

Query	Choke points	Morph-GraphQL support
Q1	1.1, 2.1, 2.2	Yes
Q2	2.1	Yes
Q3	2.2, 2.3	Yes
Q4	2.2, 2.3, 2.5	Yes
Q5	2.1, 2.2, 2.3, 2.4	Yes
Q6	2.2, 2.5	Yes
Q7	2.2, 2.5, 3.2	No
Q8	3.1, 3.3	No
Q9	2.1, 2.2, 3.1, 3.3	No
Q10	1.1, 4.1	No
Q11	2.5, 4.4	No
Q12	2.5, 4.3	No
Q13	2.1, 4.2, 4.3	No
Q14	2.1, 4.2, 4.3, 4.5	No
Q15	5.2	No
Q16	5.1	No

4.2. *Evaluation setup*

We used the LinGBM Data Generator to generate various sizes of datasets[l] (1 K, 2 K, 4 K, 8 K, 16 K, 32 K, 64 K and 128 K) and loaded them in the relational database. We created declarative mappings following the mapping guidelines provided in the benchmark. For each query template, we generated 20 queries with different instances generated randomly, as it is the default settings of the benchmark. To measure the performance, we run each query instance 5 times in cold mode, and we calculate the average for each query. We also generated the equivalent SPARQL queries for evaluating the other approaches (materialized knowledge graph and SPARQL-to-SQL). The knowledge graph is generated by Morph-RDB [17] from the datasets using the same declarative mappings. All the resources used in the evaluation are available online on our GitHub repository.[m] Figure 3 illustrates the evaluation workflow explained above. The figure shows a materialized and virtualized knowledge graphs. The materialized knowledge graph is the data transformed to RDF and stored into a triple store "LinGBM Knowledge Graph" (we use virtuoso in this case). The virtualized view does not transform the original datasets to another format (the data is stored in a RDB). It provides access to the underlying datasets via GraphQL. Once a GraphQL query is received, the GraphQL server uses GraphQL engine to translate the query into SQL, which would query the Database "LinGBM RDB", get the results, and then the results of the SQL query will be changed into the requested format according

[l]The size is defined in terms of number of products in the database (1 K means 1 thousand products).
[m]`https://github.com/oeg-upm/morph-graphql/`.

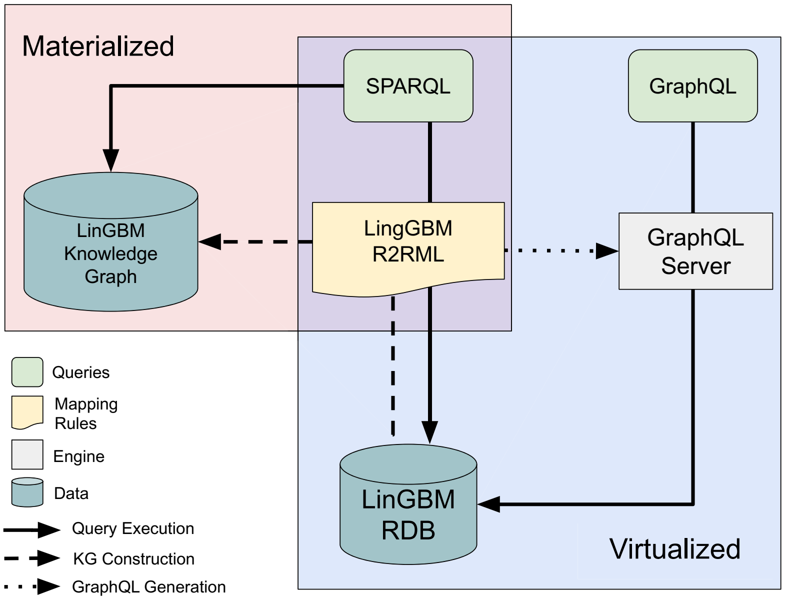


Fig. 3. **Evaluation Workflow.** We evaluated Morph-GraphQL by comparing its performance over the supported LinGBM queries against two equivalent Semantic Web approaches. The first one is the materialization of the LinGBM RDB to RDF using R2RML mappings and the second one is the translation from SPARQL-to-SQL using also the same mapping rules.

to the GraphQL query. We measure the total query execution time for the following queries:

- GraphQL queries that are translated into SQL queries and evaluated in a relational database instance database. We use Morph-GraphQL v1.0.0 to evaluate these queries.
- SPARQL queries that are evaluated over the materialized knowledge graph that is, queries are evaluated using a triple store. We use a Virtuoso v7.2.5.1 instance in this case.
- SPARQL queries that are evaluated over the virtual knowledge graph that is, queries are translated into SQL queries and evaluated by Morph-RDB v3.9.15 over the relational database instance.

The experiments were run in an Intel(R) Xeon(R) equipped with a CPU E5-2603 v3 @ 1.60 GHz 20 cores, 100 G memory with Ubuntu 16.04LTS.

4.3. *Results and discussion*

In terms of choke-points, Morph-GraphQL supported queries that belong solely to two classes: *attribute retrieval* and *relationship traversal*. Queries which belong to the classes *ordering-and-paging* and *searching-and-filtering* are not supported by Morph-GraphQL. Nonetheless, this can be addressed in a future version of Morph-GraphQL. The last class of choke points addresses *aggregations*, which has not been addressed yet in the GraphQL specification.

We show the results for the different dataset sizes in Table 2. We see that for smaller dataset sizes (1 K, 2 K, and 4 K), Morph-GraphQL outperforms the others for

Table 2. Query evaluation performance (time in seconds) over multiple sizes of the LinGBM (the number indicates the scale factor used). Execution time is a lower-is-better metric.

Engine/Queries	Q1	Q2	Q3	Q4	Q5	Q6	Geometric mean
LinkGBM 1 K							
Morph-GraphQL	0.103	0.146	0.005	0.118	0.237	0.079	0.075
Morph-RDB	0.168	0.081	0.201	0.221	0.296	0.089	0.159
Virtuoso	0.182	0.017	0.091	0.079	1.204	0.131	0.124
LinkGBM 2 K							
Morph-GraphQL	0.183	0.200	0.006	0.171	0.397	0.071	0.100
Morph-RDB	0.167	0.085	0.203	0.224	0.308	0.088	0.161
Virtuoso	0.096	0.025	0.057	0.068	1.291	0.073	0.098
LinkGBM 4 K							
Morph-GraphQL	0.314	0.316	0.005	0.311	0.799	0.096	0.151
Morph-RDB	0.171	0.083	0.199	0.223	0.318	0.089	0.161
Virtuoso	0.109	0.035	0.059	0.078	12.293	0.101	0.167
LinkGBM 8 K							
Morph-GraphQL	0.597	0.644	0.004	0.625	1.363	0.096	0.228
Morph-RDB	0.178	0.080	0.196	0.225	0.323	0.090	0.162
Virtuoso	0.096	0.070	0.064	0.069	2.142	0.097	0.135
LinkGBM 16 K							
Morph-GraphQL	1.121	1.408	0.005	1.293	2.776	0.104	0.376
Morph-RDB	0.167	0.083	0.205	0.222	0.322	0.089	0.162
Virtuoso	0.100	0.122	0.057	0.073	1.412	0.090	0.137
LinkGBM 32 K							
Morph-GraphQL	2.635	2.884	0.005	2.543	6.086	0.130	0.644
Morph-RDB	0.173	0.085	0.199	0.220	0.323	0.089	0.163
Virtuoso	0.108	0.274	0.069	0.085	1.591	0.122	0.179
LinkGBM 64 K							
Morph-GraphQL	5.157	5.940	0.005	5.114	11.065	0.147	1.050
Morph-RDB	0.177	0.085	0.199	0.224	0.325	0.090	0.164
Virtuoso	0.116	0.417	0.057	0.091	1.666	0.102	0.187
LinkGBM 128 K							
Morph-GraphQL	8.806	9.552	0.005	8.437	22.453	0.152	1.526
Morph-RDB	0.172	0.084	0.199	0.224	0.324	0.090	0.163
Virtuoso	0.120	0.381	0.058	0.090	1.613	0.115	0.188

the majority of the queries. The main reason of these results for smaller dataset sizes is because of the overhead (for translating queries from SPARQL to SQL and optimizing the resulting SQL) in Morph-RDB has a negative impact in the total performance of the query execution. Morph-GraphQL needs less time translating the query because it is relatively more simple to translate GraphQL to SQL compared to SPARQL to SQL. In most of the cases for these dataset sizes, Virtuoso needs more time than the other two systems due to the absence of indexes in RDF, which has a negative impact depending on the features of the query [18]. For bigger datasets, SPARQL to SQL optimizations [17] implemented in Morph-RDB pays off the translation time and gives better impact over the query execution process, outperforming Morph-GraphQL, which hints that the optimizations in the query translation from GraphQL to SQL can still be improved in order to query big datasets (32 K, 64 K, 128 K). When we consider the whole set of queries and calculate the geometric mean of the results, we notice that Morph-GraphQL outperforms the others in some of the dataset sizes because of its fast execution time for the queries Q3 and Q6. Analyzing the queries individually, we can observe that for all the engines, Q5 is the most costly one due to the number of nested queries that it consists.

5. Related Work

The GraphQL-LD specification is proposed in [19], where the authors include a context to GraphQL queries, similar to what is proposed in JSON-LD [20]. The goal of this work is to translate GraphQL queries into SPARQL queries for querying RDF interfaces and provide a more friendly interface for developers. HyperGraphQL[n] used an intermediary service to use a GraphQL interface for querying and serving linked data on the Web. For the in-depth comparison between GraphQL-LD and HyperGraphQL, we refer the reader to [21].

Ontop [22] proposed several semantic optimization techniques to generate efficient SQL queries resulting from the translation of SPARQL queries taking into account R2RML mappings. Morph-RDB [17] presented an R2RML-based SPARQL to SQL query translation based on the approached proposed by Chebotko *et al.* [13]. The approach that we proposed in this paper can be considered as an alternative to query translation technique as it allows the answering of GraphQL queries over local datasets without materializing them, by translating declarative mappings into GraphQL resolvers and delegate the query evaluation to GraphQL engines.

Table 3. Summary of approaches.

Proposal	Input	Output
Chebotko *et al.*	SPARQL	SQL
Morph-RDB	SPARQL + R2RML	SQL
ontop	SPARQL + R2RML	SQL
GraphQL-LD	GraphQL (Query) + Context	SPARQL
Morph-GraphQL	R2RML or RML	GraphQL Server

Another relevant work is [23], in which the authors analyze and formalize the semantics and the complexity of GraphQL. Their theoretical study can be used for

[n] https://www.hypergraphql.org/.

further analysis of the query language while their technical contributions help GraphQL developers to implement more robust interfaces for the Web.

In [24], we introduced the concept of *mapping translation* and its two properties: data and query result preservation properties. In summary, mapping translation is a process where an instance of one mapping language is translated into an instance of another mapping language. We iterated several cases in which mapping translation may be useful [10, 25–27]. Our proposed work in this paper can be seen as an application of mapping translation in which we consider GraphQL resolvers as a mapping language (that specifies relationships between the source schema and the exposed GraphQL schema).

Finally, systems and approaches that help the creation of the mapping rules are also important and relevant for our proposal. There are that engines that automatically build a draft of the mapping document exploiting the information of the relational database schema. Examples of these engines are MIRROR [28], AutoMap4OBDA [29], KARMA [12] and BootOX [30]. Other systems are focused in facilitating the knowledge engineer with the creation of these rules such as RMLEditor [11] and Mapeathor [25].

6. Conclusions and Future Work

In this paper, we have presented an approach to generate GraphQL resolvers from R2RML mappings together with its corresponding implementation, Morph-GraphQL. Note that we do not aim to replace the traditional approach of generating GraphQL schema/resolvers manually, but we position this approach as a supplementary approach. This is to say that this approach allows domain experts to use the generated schema and resolvers as the initial proof of concept that can be used to query datasets without the need for software engineers to develop a full-fledged GraphQL server. Software engineers may also benefit from our approach as they may also use Morph-GraphQL to generate the initial version of a GraphQL server instead of building it from scratch.

In the future, we plan to integrate Morph-GraphQL with Mappingpedia [31], a repository for R2RML mappings, to make it easier for users to share their mappings and explore existing mappings. TADA [32] proposes an approach to generate mappings for tabular datasets automatically. We will also integrate TADA with Morph-GraphQL. We also plan to evaluate our approach comparing the time taken by a domain expert to generate R2RML mappings and a software engineer programming a GraphQL resolver.

Acknowledgments

We would like to acknowledge David Moreno and Álvaro Sainz for helping in the development of Morph-GraphQL and Daniel Doña for helping in the setup of the performed experiments. The work presented in this paper is supported by the

Spanish Ministerio de Economía, Industria y Competitividad and EU FEDER funds under the DATOS 4.0: RETOS Y SOLUCIONES - UPM Spanish National Project (TIN2016-78011-C4-4-R) and by an FPI Grant (BES-2017-082511).

References

1. M. Bryant, GraphQL for archival metadata: An overview of the EHRI GraphQL API, in *IEEE Int. Conf. Big Data*, 2017, pp. 2225–2230.
2. M. Vogel, S. Weber and C. Zirpins, Experiences on migrating restful web services to GraphQL, in *Int. Conf. Service-Oriented Computing*, 2017, pp. 283–295.
3. S. K. Mukhiya, F. Rabbi, V. K. I. Pun, A. Rutle and Y. Lamo, A GraphQL approach to healthcare information exchange with HL7 FHIR, *Proc. Comput. Sci.* **160** (2019) 338–345.
4. Facebook, Inc., GraphQL, 2018, `https://facebook.github.io/graphql/June2018/`.
5. S. Das, S. Sundara and R. Cyganiak, R2RML: RDB to RDF Mapping Language, 2018, https://www.w3.org/TR/r2rml/.
6. A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens and R. Van de Walle, RML: A generic language for integrated RDF mappings of heterogeneous data, in *Proc. 7th Workshop on Linked Data on the Web*, 2014, http://ceur-ws.org/Vol-1184/.
7. A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini and R. Rosati, Linking data to ontologies, *Journal on Data Semantics X* (2008) 133–173.
8. F. Michel, L. Djimenou, C. Faron-Zucker and J. Montagnat, Translation of relational and non-relational databases into RDF with xR2RML, in *11th Int. Conf. Web Information Systems and Technologies*, 2015, pp. 443–454.
9. J. Slepicka, C. Yin, P. A. Szekely and C. A. Knoblock, KR2RML: An alternative interpretation of R2RML for heterogenous sources, in *Proc. Consuming Linked Data*, 2015, http://ceur-ws.org/Vol-1426/.
10. D. Chaves-Fraga, F. Priyatna, I. Perez-Santana and O. Corcho, Virtual statistics knowledge graph generation from CSV files, in *Emerging Topics in Semantic Technologies: ISWC 2018 Satellite Events*, Studies on the Semantic Web, Vol. 36, 2018, pp. 235–244.
11. P. Heyvaert, A. Dimou, A.-L. Herregodts, R. Verborgh, D. Schuurman, E. Mannens and R. Van de Walle, Rmleditor: A graph-based mapping editor for linked data mappings, in *European Semantic Web Conference*, 2016, pp. 709–723.
12. C. A. Knoblock and P. Szekely, Exploiting semantics for big data integration, *AI Magazine* **36**(1) (2015) 25–38.
13. A. Chebotko, S. Lu and F. Fotouhi, Semantics preserving SPARQL-to-SQL translation, *Data Knowl. Eng.* **68**(10) (2009) 973–1000.
14. F. Priyatna, D. Chaves-Fraga, A. Alobaid and O. Corcho, Morph-GraphQL: GraphQL servers generation from R2RML mappings, in *Proc. 31st Int. Conf. Software Engineering and Knowledge Engineering*, 2019.
15. O. Hartig, S. Cheng and L. Lindqvist, Linköping GraphQL Benchmark (LinGBM) kernel description, 2019, https://github.com/LiUGraphQL/LinGBM.
16. C. Bizer and A. Schultz, The Berlin SPARQL benchmark, *Int. J. Semantic Web Inf. Syst.* **5**(2) (2009) 1–24.
17. F. Priyatna, O. Corcho and J. Sequeda, Formalisation and experiences of R2RML-based SPARQL to SQL query translation using Morph, in *Proc. 23rd Int. Conf. World Wide Web*, 2014, pp. 479–490.

18. K. M. Endris, P. D. Rohde, M.-E. Vidal and S. Auer, Ontario: Federated query processing against a semantic data lake, in *Int. Conf. Database and Expert Systems Applications*, 2019, pp. 379–395.
19. R. Taelman, M. Vander Sande and R. Verborgh, GraphQL-LD: Linked Data Querying with GraphQL, in *17th Int. Semantic Web Conference*, 2018, http://ceur-ws.org/Vol-2180/.
20. M. Sporny, D. Longley, G. Kellogg, M. Lanthaler and N. Lindström, JSON-LD 1.0, *W3C Recommendation* **16** (2014) 41.
21. J. Werbrouck, M. Senthilvel, J. Beetz, P. Bourreau and L. Van Berlo, Semantic query languages for knowledge-based web services in a construction context, in *Proc. 26th Int. Workshop on Intelligent Computing in Engineering*, 2019, http://ceur-ws.org/Vol-2394/.
22. D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro and G. Xiao, Ontop: Answering SPARQL queries over relational databases, *Semantic Web* **8**(3) (2017) 471–487.
23. O. Hartig and J. Pérez, Semantics and complexity of GraphQL, in *Proc. World Wide Web Conf.*, International World Wide Web Conferences Steering Committee, 2018, pp. 1155–1164.
24. O. Corcho, F. Priyatna and D. Chaves-Fraga, Towards a new generation of ontology based data access, *Semantic Web J.* **11**(1) (2019) 153–160.
25. A. Iglesias-Molina, D. Chaves-Fraga, F. Priyatna and O. Corcho, Towards the definition of a language-independent mapping template for knowledge graph creation, in *Proc. Third Int. Workshop on Capturing Scientific Knowledge*, 2019, http://ceur-ws.org/Vol-2526/.
26. P. Heyvaert, B. De Meester, A. Dimou and R. Verborgh, Declarative rules for linked data generation at your fingertips!, in *Proc. 15th ESWC: Posters and Demos*, 2018.
27. D. Chaves-Fraga, E. Ruckhaus, F. Priyatna, M.-E. Vidal and O. Corcho, Enhancing OBDA query translation over tabular data with Morph-CSV, preprint, 2020, arXiv:2001.09052.
28. L. F. de Medeiros, F. Priyatna and O. Corcho, Mirror: Automatic R2RML mapping generation from relational databases, in *Int. Conf. Web Engineering*, 2015, pp. 326–343.
29. Á. Sicilia and G. Nemirovski, Automap4obda: Automated generation of R2RML mappings for obda, in *European Knowledge Acquisition Workshop*, 2016, pp. 577–592.
30. E. Jiménez-Ruiz, E. Kharlamov, D. Zheleznyakov, I. Horrocks, C. Pinkel, M. G. Skjæveland, E. Thorstensen and J. Mora, BootOX: Practical mapping of RDBS to OWL 2, in *Int. Semantic Web Conf.*, 2015, pp. 113–132.
31. F. Priyatna, E. Ruckhaus, N. Mihindukulasooriya, Ó. Corcho and N. Saturno, Mappingpedia: A collaborative environment for R2RML mappings, in *European Semantic Web Conference*, 2017, pp. 114–119.
32. A. Alobaid and O. Corcho, Fuzzy semantic labeling of semi-structured numerical datasets, in *European Knowledge Acquisition Workshop*, 2018, pp. 19–33.