morph-GraphQL: GraphQL Servers Generation from R2RML Mappings (SESE)*

1st Freddy Priyatna
Ontology Engineering Group
Universidad Politecnica de Madrid
Madrid, Spain
fpriyatna@fi.upm.es

2nd David Chaves-Fraga
Ontology Engineering Group
Universidad Politecnica de Madrid
Madrid, Spain
dchaves@fi.upm.es

3rd Ahmad Alobaid
Ontology Engineering Group
Universidad Politecnica de Madrid
Madrid, Spain
aalobaid@fi.upm.es

4th Oscar Corcho
Ontology Engineering Group
Universidad Politecnica de Madrid
Madrid, Spain
ocorcho@fi.upm.es

***Abstract*—REST has become in the last decade the most common manner to provide web services, yet it was not originally designed to handle typical modern applications (e.g., mobile apps). GraphQL was released publicly in 2015 and since then has gained momentum as an alternative approach to REST. However, generating and maintaining GraphQL resolvers is not easy. First, a domain expert has to analyse 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 that generates GraphQL resolvers from declarative mappings specification in the W3C Recommendation R2RML, hence, can be used both by a domain expert as without the need to involve software developers to implement the resolvers, and by software developers as the initial version of the resolvers to be implemented. Our approach is implemented in morph-GraphQL.**

***Index Terms*—GraphQL, R2RML, OBDA**

I. Introduction

Introduced in 2000, Representational State Transfer (REST) has become the most common manner to provide web services in the last few years. 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*. Over-fetching refers to the situation in which a REST endpoint returns more data than what is required by the developer. 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. 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 [6], 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. 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 implemented in a programming language that are responsible to translate GraphQL queries into queries supported by the underlying datasets (e.g. GraphQL to SQL). GraphQL is supported by multiple GraphQL engines for major programming languages (e.g. JavaScript, Python, Java, Golang, Ruby). In addition to the above mentioned frameworks, query planning tools have been developed in order to translate GraphQL queries into other query languages (e.g. dataloader[1], joinmonster[2]).

[1]https://github.com/facebook/dataloader
[2]https://join-monster.readthedocs.io/en/latest/

DOI reference number: 10.18293/SEKE2019-055

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 will analyse 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 (e.g. the domain expert may use a spreadsheet, which is not necessarily easy to understand by another domain expert).
2) A software developer then implements 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 is becoming more difficult even for a standard-sized dataset which typically contains more than a handful tables and hundreds of properties. This situation is even worse if the underlying dataset evolves considering that the corresponding resolvers have to be updated as well. GraphQL resolvers may not be easily understood by new developers who were not involved in the initial version thus bringing the possibility of introducing errors.

In this paper we propose the use of W3C R2RML [4] to specify the mapping rules that relate the source datasets and the GraphQL schema. The use of R2RML mappings is based on the idea that the use of a standard mapping language would facilitate better understanding of the mapping from the underlying data source and the exposed GraphQL schema. Furthermore, they also allow for better maintainability as R2RML mappings are declarative and independent from any programming language. Our main contribution in this paper is, taking the advantage that R2RML mappings are declarative, an approach to translate R2RML mappings to JavaScript-based GraphQL resolvers.

The rest of the paper is structured as follows: in section II we review R2RML and GraphQL and in section III we describe our approach on translating R2RML mappings to GraphQL resolvers. In section IV we present the queries, based on the example provided in the reference implementation[3], that we use to test our implementations. Finally, related work and conclusion are presented in sections V and VI.

II. Background

In this section we provide some background on two of the underlying technologies that we will use: GraphQL and R2RML. We use the example 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 is shown in Figure 1.

A. GraphQL

GraphQL is a specification that provides a unified view for accessing heterogeneous datasets using its query language.

[3]https://github.com/graphql/graphql-js/

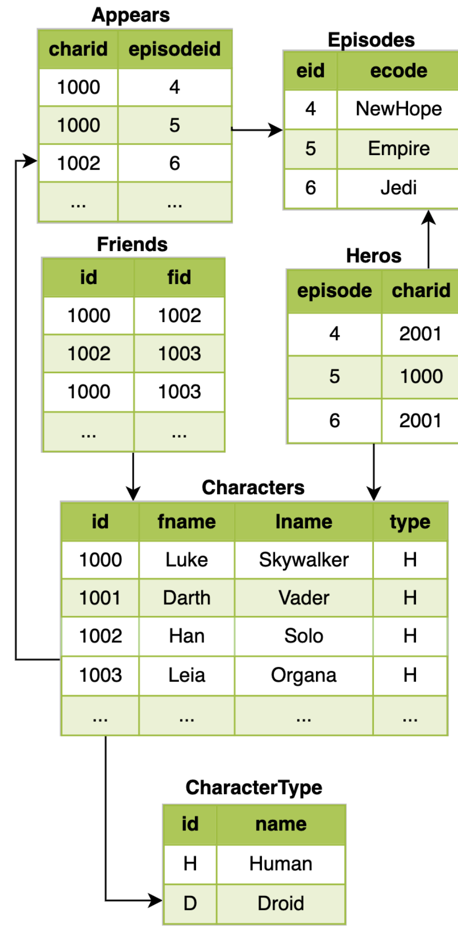

Fig. 1. Tables used in the Star Wars example, inspired by the example provided in the reference implementation

Besides the query language, the specification defines how a GraphQL server may be implemented for allowing the 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 (query root and type).

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

type Episode {
    identifier:String
    code:String
}
```

Listing 1. GraphQL Schema for type Episode

A GraphQL schema specifies all the available types and their properties. For example, in Listing 1 we can see that the schema for the `Episode` type together with its two fields: `identifier` and `code`.

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`).

Listing 2 shows a possible JavaScript implementation of the resolver for the `Episode` type. This part of the code is responsible for filtering out instances based on the fields `identifier` or `code`.

```
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 2. GraphQL Resolver for Type Episode

B. R2RML

The W3C R2RML Recommendation (September 2012) allows users to specify rules for transforming relational database content into an *R2RML output dataset*, the resulting graph from applying R2RML mappings. The transformation rules are defined in an R2RML mapping document that contains a set of Triples Map (`rr:TriplesMap`). Triples Maps are used to generate RDF triples from logical tables. A Triples Map consists of:

- a Logical Table (`rr:LogicalTable`) that specifies the source relational table/view.
- 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, respectively. If a join with another Triples Map is needed, a Reference Object Map (`rr:RefObjectMap`) may be specified.

A Term Map (`rr:TermMap`) is either Subject Map, Predicate Map, and 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 column-valued map (`rr:column`), or a template-valued map (`rr:template`). Furthermore, additional information such as datatype (`rr:datatype`) can also be attached to Term Maps.

In Listing 3 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.

```
<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 3. R2RML Mapping for Episode

III. Approach

Our approach (Figure 2) generates GraphQL servers from R2RML mappings. Hence, mappings can be created by a domain expert in a declarative language, without the need for programming skills, while benefiting from the wide range of tools available for GraphQL in order to access data stored in tabular format (i.e., RDB or CSV). The approach consists of the following steps: 1) the generation of a SQL query, 2) the generation of schema and 3) the generation of resolvers, from each Triples Map defined in the mapping document.

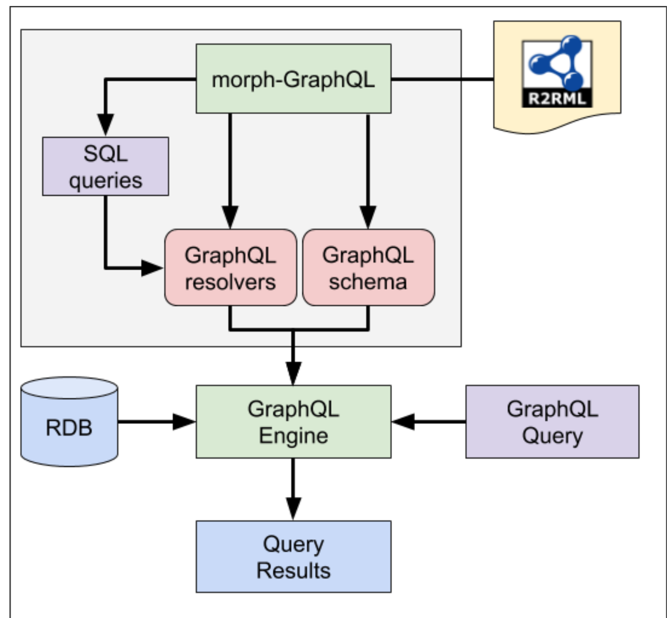

Fig. 2. **morph-GraphQL workflow.** morph-GraphQL receives R2RML mappings and generates SQL queries to be used in GraphQL resolvers. Then, it generates a GraphQL server (schema + resolvers) that can be used by a GraphQL engine to evaluate queries over the RDB data.

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

- $getConstant(TermMap)$ retrieves the constant `c` in the constant-value term map $TermMap$ = `rr:constant "c"`.
- $getColumn(TermMap)$ retrieves the column `col` in the column-value term map $TermMap$ = `rr:column "col"`.
- $templateToSQL(TemplateValue)$ converts a template-value term map into an SQL expression. For example, given the term map `rr:template "ex.com/episode/{eid}"` as the input, this function may return `"ex.com/episode/" || eid` or `CONCAT("ex.com/episode/{eid}", eid)`, depending on the database system being used.
- $transDataType(xsdDataType)$ that given an XSD Data Type return the corresponding GraphQL type. For example, $transDataType(\text{"}xsd:string\text{"})$ returns `String`.
- `join(objs, separator)` that joins a collection of objects $objs$ into a string with the separator $separator$. For example, $join([1, 2, 3], \text{"}AND\text{"})$ returns `"1 AND 2 AND 3"`.

A. Generating SQL Queries

We present here a set of translation functions that translates a triples map into the corresponding SQL query to be used in GraphQL resolvers. This set of functions is adapted from the work presented in [3], which is used to translate SPARQL queries into SQL queries without the presence of R2RML mappings.

- $\alpha(TriplesMap)$ returns a set of logical tables associated with the triples map $TriplesMap$, which is the logical table associated to the triples map $TriplesMap$ and additionally all the parent tables if $TriplesMap$ contains Referenced Object Maps.
- $\beta(TermMap)$ that given a term map $TermMap$ returns the corresponding SQL expression, that is:
  - $getConstant(TermMap)$ if $TermMap$ is a constant-value map
  - $getColumn(TermMap)$ if $TermMap$ is a column-value map
  - $templateToSQL()$ if $TermMap$ is a template-value map.
- $alias(TermMap)$ generates a unique alias to be used in the generation of SQL statement
- $genPRSQL(TriplesMap)$ generates a SQL expression which projects the relevant SQL expresions of a triples map $TriplesMap$ (i.e., β of Subject Map and all Object Maps) together with their aliases.
- $genCondSQL(TriplesMap)$ generates a SQL expression which is evaluted to true if they match the arguments passed in the resolver functions and additionaly the join conditions if $TriplesMap$ contains Referenced Object Maps.
- finally, $trans(TM)$ = "SELECT $genPRSQL(TM)$ FROM $\alpha(TM)$ WHERE $genCondSQL(TM)$" translates a triples map into the corresponding SQL query.

Example Given Listing 3 as the input, $trans$ generates the SQL query that can be seen in variable sql in Listing 2.

B. Generating Schema

Algorithm 1 generates a GraphQL schema from a Triple Map. It simply generates a GraphQL type $MappedClass$, where $MappedClass$ is the class specified in the Subject Map of the Triples Map. The fields of the $MappedClass$ are `identifier` and all the mapped predicates in the Predicate Object Maps of the Triples Map. The datatype of the fields are the results of function $transDataType$, which returns the corresponding GraphQL type from the datatype specified in the Object Maps of the Triples Map.

Algorithm 1 GenerateSchema(TriplesMap)

```
SM = TriplesMap.getSubjectMap()
MappedClass = SM.getMappedClass()
POMS = TriplesMap.getPredicateObjectMaps()
Result = "type MappedClass {"
Result += "identifier:String"
for all POM ← POMS do
    PM = POM.getPredicateMap()
    OM = POM.getObjectMap()
    PMConstant = PM.getConstant()
    DataType = transDataType(OM.getDataType)
    Result += "PMConstant:Datatype"
end for
Result += "}"
return Result
```

Example Given Listing 3 as the input, Algorithm 1 generates GraphQL Type `Episode` that can be seen in Listing 1.

C. Generating Resolvers

Algorithm 2 generates a GraphQL resolver from a TriplesMap. As for the name of the resolver, we opt for `list`$MappedClass$, that is, a Triples Map whose mapped class is `Episode` will generate a resolver `listEpisode`. This resolver will use the SQL query generated from section III-A, execute the SQL query on the underlying database engine, and then generate the corresponding instances by calling the constructor of Type $MappedClass$.

Example Given Listing 3 as the input, Algorithm 2 generates resolvers that can be seen in Listing 2.

IV. Implementation and Queries

As of the time of writing, we have implemented **morph-GraphQL**[4], an open source tool to translate R2RML mappings into Javascript-based GraphQL resolvers. Currently, it

[4]https://github.com/oeg-upm/morph-graphql, deployed at http://graphql.morph.oeg-upm.net

Algorithm 2 GenerateQueryRoot(TriplesMap)

```
SM = TriplesMap.getSubjectMap()
MappedClass = SM.getMappedClass()
POMS = TriplesMap.getPredicateObjectMaps()
PMSConstants = [identifier]
for all POM ← POMS do
  PM = POM.getPredicateMap()
  PMConstant = PM.getConstant()
  PMSConstants.push(PMConstant)
end for
Result = ""
Result += "listMappedClass:
functions(PMSConstants.join(",")) {"
Result += "sql = trans(TriplesMap)"
Result += "rows = db.all(sql)"
Result += "allInstances = []"
for all row ← rows do
  args = []
  for all POM ← POMS do
    PM ← POM.getPredicateMapping()
    PMConstant ← PM.getConstant()
    args.push(row.[alias(PMConstant)])
  end for
  Result += "instance = new
  MappedClass(args)"
  Result += "allInstances.push(instance)"
  Result += "return allInstances"
end for
Result += "}"
return Result
```

is able to generate resolvers for accessing tabular datasets, such as RDB or CSV files. We use the JoinMonster library[5] to generate efficient SQL queries when joins are needed.

Due to the recent emergence of GraphQL, and as far as we are aware of, there has not been any standard benchmark or test-case proposed for evaluating the conformance and performance of a GraphQL-compliant framework. In order to test our approach, we use a set of GraphQL queries with various degrees of complexity proposed in the example of the reference implementation. First, we serialize the Star Wars instance data in a tabular format (Figure 1) and then generate its corresponding R2RML mapping document. Then we evaluate the queries in Table I. All the information about the dataset, mapping and queries and their results is available online[6]. Additionally, a GraphQL server that is ready to answer those queries has been also deployed[7].

The initial version of morph-GraphQL presented in this paper shows that an R2RML mapping document can be used to generate automatically GraphQL resolvers. Besides, the implementation of the resolvers is able to cover various levels of query complexity so that it can be used as a tool for accessing heterogenous datasets via GraphQL queries.

TABLE I
STAR WARS QUERIES

No	Description	Tables Involved
Q1	Query the hero of every episode	heroes, episodes, characters
Q2	Query for the id and friends of R2-D2	characters, friends
Q3	Query for Luke Skywalker directly, using his ID	characters
Q4	Query for both Luke and Leia	characters
Q5	Verify that R2-D2 is a droid	characters, types
Q6	Verify that the hero of episode Empire is a human	heroes, characters, types, episodes

V. RELATED WORK

Several works are at the intersection of GraphQL and Ontology-Based Data Access (OBDA) [9]. In OBDA, ontologies are used as a global view over heterogeneous local datasets and the relationship between them is specified by mappings. R2RML is an example of declarative OBDA mappings whose focus is the generation of ontology instances from relational databases. Other related declarative proposals are: RML [5] (to deal with CSVs, JSON and XML data sources), xR2RML [8] (to deal with MongoDB), KR2RML [12] (to deal with nested data) or RMLC-Iterator (for statistical CSV files) [2]. Two techniques for answering queries over the global schema are: data translation and query translation. In data-translation, a set of mapping rules is used to generate the instances of the global schema and then those instances are materialised in a triple store so that queries posed over the global schema can be evaluated by the triple store. 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, thus eliminating the need of materialisation.

The GraphQL-LD specification is proposed in [14], where the authors include a context to GraphQL queries, similar as it is proposed in JSON-LD [13]. The goal of this work is to translate GraphQL queries to SPARQL queries for querying RDF interfaces and provide a more friendly interface for the developers. Ontop [1] proposed several semantic optimisation techniques to generate efficient SQL queries resulting from the translation of SPARQL queries taking into account R2RML mappings. morph-RDB [10] presented an R2RML-based SPARQL to SQL query translation based on the approached proposed by Chebotko et. al [3]. The approach that we proposed in the paper can be considered as a query translation technique as it allows the answering of GraphQL queries over local datasets without materialising them, by translating R2RML mappings into GraphQL resolvers and delegate the query evaluation to GraphQL engines. Note, however, unlike previous approaches that take a query as their input in their run-time, morph-GraphQL is compile-time, in sense that the generation of SQL and GraphQL resolvers

[5]https://join-monster.readthedocs.io
[6]https://github.com/oeg-upm/morph-graphql/wiki/Example-Star-Wars
[7]http://starwars.graphql.oeg-upm.net/graphql

TABLE II
SUMMARY OF APPROACHES

Proposal	Input	Output	Type
Chebotko et al	SPARQL	SQL	run time
morph-RDB	SPARQL + R2RML	SQL	run time
ontop	SPARQL + R2RML	SQL	run time
GraphQL-LD	GraphQL	SPARQL	run time
morph-GraphQL	R2RML	SQL + GraphQL (Schema & Resolvers)	compile time

are only executed once. We summarise the aforementioned approaches in Table II.

Another relevant work is [7], in which the authors analyse and formalise the semantics and the complexity of GraphQL. Their theoretical study can be used for further analysis of the query language while their technical contributions help GraphQL developers to implement more robust interfaces for the web.

VI. CONCLUSION

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 supplementary approach. This is to say, 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 support more programming languages (e.g. Java) and more data formats (e.g. JSON) and integrate morph-GraphQL with Mappingpedia [11], a repository for R2RML mappings. 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.

ACKNOWLEDGMENT

We are thankful to Nandana Mihindukulasooriya, Anastasia Dimou, Ben de Meester and Pieter Heyvaert, who helped us in the identifying the main contributions of our approach. 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] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering sparql queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.

[2] David Chaves-Fraga, Freddy Priyatna, Idafen Perez-Santana, and Oscar Corcho. Virtual statistics knowledge graph generation from CSV files. In *Emerging Topics in Semantic Technologies: ISWC 2018 Satellite Events*, volume 36 of *Studies on the Semantic Web*, pages 235–244. IOS Press, 2018.

[3] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering*, 68(10):973–1000, 2009.

[4] Souripriya Das, Seema Sundara, and Richard Cyganiak. R2RML: RDB to RDF Mapping Language. https://www.w3.org/TR/r2rml/. Accessed: 2018-12-07.

[5] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *LDOW*, 2014.

[6] Facebook, Inc. GraphQL. https://facebook.github.io/graphql/June2018/, 2018. Accessed: 2018-12-07.

[7] Olaf Hartig and Jorge Pérez. Semantics and complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1155–1164. International World Wide Web Conferences Steering Committee, 2018.

[8] Franck Michel, Loïc Djimenou, Catherine Faron-Zucker, and Johan Montagnat. Translation of relational and non-relational databases into RDF with xR2RML. In *11th International Confenrence on Web Information Systems and Technologies (WEBIST'15)*, pages 443–454, 2015.

[9] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In *Journal on data semantics X*, pages 133–173. Springer, 2008.

[10] Freddy Priyatna, Oscar Corcho, and Juan Sequeda. Formalisation and experiences of R2RML-based SPARQL to SQL query translation using morph. In *Proceedings of the 23rd international conference on World wide web*, pages 479–490. ACM, 2014.

[11] Freddy Priyatna, Edna Ruckhaus, Nandana Mihindukulasooriya, Óscar Corcho, and Nelson Saturno. Mappingpedia: A collaborative environment for R2RML mappings. In *European Semantic Web Conference*, pages 114–119. Springer, 2017.

[12] Jason Slepicka, Chengye Yin, Pedro A Szekely, and Craig A Knoblock. KR2RML: An alternative interpretation of r2rml for heterogenous sources. In *COLD*, 2015.

[13] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. Json-ld 1.0. *W3C Recommendation*, 16:41, 2014.

[14] Ruben Taelman, Miel Vander Sande, and Ruben Verborgh. GraphQL-LD: Linked Data Querying with GraphQL. In *ISWC2018, the 17th International Semantic Web Conference*, 2018.