Ontology-Based GraphQL Server Generation for Data Access and Data Integration

Huanyu Li [a,b,*], Olaf Hartig [a], Rickard Armiento [b,c] and Patrick Lambrix [a,b,d]
[a] *Department of Computer Science, Linköping University, Sweden*
E-mails: huanyu.li@liu.se, olaf.hartig@liu.se, patrick.lambrix@liu.se
[b] *The Swedish e-Science Research Centre, Linköping University, Sweden*
E-mails: huanyu.li@liu.se, rickard.armiento@liu.se, patrick.lambrix@liu.se
[c] *Department of Physics, Chemistry and Biology, Linköping University, Sweden*
E-mail: rickard.armiento@liu.se
[d] *Department of Building Engineering, Energy Systems and Sustainability Science, University of Gävle, Sweden*
E-mail: patrick.lambrix@liu.se

Abstract. In a GraphQL Web API, a so-called GraphQL schema defines the types of data objects that can be queried, and so-called resolver functions are responsible for fetching the relevant data from underlying data sources. Thus, we can expect to use GraphQL not only for data access but also for data integration, if the GraphQL schema reflects the semantics of data from multiple data sources and the resolver functions can obtain data from these data sources and structure the data according to the schema. However, there does not exist a semantics-aware approach to employ GraphQL for data integration. Furthermore, there are no formal methods for defining a GraphQL API based on an ontology. In this paper, we introduce a framework for using GraphQL in which a global domain ontology informs the generation of a GraphQL server that answers requests by querying heterogeneous data sources. The core of this framework consists of an algorithm to generate a GraphQL schema based on an ontology and a generic resolver function based on semantic mappings. We provide a prototype, OBG-gen, of this framework, and we evaluate our approach over a real-world data integration scenario in the materials design domain and two synthetic benchmark scenarios (Linköping GraphQL Benchmark and GTFS-Madrid-Bench). The experimental results of our evaluation indicate that: (i) our approach is feasible to generate GraphQL servers for data access and integration over heterogeneous data sources, thus avoiding a manual construction of GraphQL servers, and (ii) our data access and integration approach is general and applicable to different domains where data is shared or queried via different ways.

Keywords: Data Integration, Ontology, GraphQL

1. Introduction

GraphQL is a conceptual framework to build APIs for Web and mobile applications [1]. It was publicly released by Facebook in 2015 and, since then, the GraphQL ecosystem[1] has grown tremendously in terms of libraries[2] supporting different programming languages (such as Python, Java and JavaScript), tools (such as Apollo and GraphiQL), and adopters (such as Airbnb, IBM and Twitter). The framework introduces the notion of a schema. Such a schema of a GraphQL API contains type definitions which specify what data objects can be retrieved from

*Corresponding author. E-mail: huanyu.li@liu.se.
[1] https://landscape.graphql.org
[2] https://graphql.org/code/

the API. The framework also contains a form of query language for expressing such data retrieval requests. Listing 1 depicts a GraphQL schema example. Based on this schema, Figure 3 shows a GraphQL query example and corresponding query result. A third component of GraphQL are resolver functions, which are typically used for executing GraphQL queries in a GraphQL server. That is, such resolvers specify—in terms of program code—how data related to the various elements of a GraphQL schema has to be fetched from underlying data sources.

GraphQL could be used to integrate data from different data sources by building a GraphQL server over these sources, in which the GraphQL schema provides a view over data from multiple sources, and the resolver functions contain implementations for accessing multiple sources. However, a semantics-aware approach to employing GraphQL for data integration does not exist. The approaches in [2] and [3] introduce how to use GraphQL for data federation. However, they are not semantics-aware. The semantics of data are not explicit in a machine-processable form which means the developer needs to write program code (i.e., resolver functions) to populate the various elements of a GraphQL schema. Furthermore, there are no formal methods for defining a GraphQL schema. The developers have to define a GraphQL schema manually. The aim of this paper is to provide a semantics-aware approach to employ GraphQL for data integration, with formal methods to generate the GraphQL server.

Research problem: This paper focuses on the following question: How can ontologies be leveraged to generate GraphQL APIs for semantics-aware data access and data integration?

Contributions: To address the research question, we propose a framework for GraphQL-based data access and integration in which an ontology drives the generation of a GraphQL server.[3] More specifically, given an ontology as an integrated view of data from multiple data sources, the first contribution is that we propose and implement a method for generating a GraphQL schema based on this ontology such that the schema becomes a view of the data to be integrated. Then, the second contribution is that we present a generic approach to create a GraphQL server that is capable to get data from the corresponding data sources by relying on semantic mappings that use the ontology.

The remainder of the paper is organized as follows. We provide the relevant background regarding ontologies, description logics, data integration and GraphQL in Section 2. We outline the GraphQL-based framework in Section 3 and elaborate on the implementation of this framework in Sections 4 and 5. Section 6 introduces related work. Section 7 presents an evaluation based on a real-world data integration scenario in the materials design domain and evaluations based on two synthetic benchmark scenarios, the Linköping GraphQL Benchmark (LinGBM) and GTFS-Madrid-Bench. Finally in Section 8, we present concluding remarks and directions for future work.

2. Background

This section provides background information on ontologies, description logics, data integration and GraphQL.

2.1. Ontologies and Description Logics

The term *ontology* originates in philosophy, in which it is the science of what is, of the kinds and structures of objects, properties, and relationships in every area of reality [4, 5]. Ontologies can be viewed, intuitively, as defining the terms, relations, and rules that combine these terms and relations in a domain of interest [6]. Through ontologies, people and organizations are able to communicate by establishing a common terminology. They provide the basis for interoperability between systems and are applicable as an index to a repository of information as well as a query model and a navigation model for data sources. Moreover, they are often used as a foundation for integrating data sources, thereby alleviating the heterogeneity issue. The benefits of using ontologies are their improved reusability, share-ability and portability across platforms, as well as their increased maintainability and reliability. On the whole, ontologies allow a field to be better understood and allow information in that field to be handled much more effectively and efficiently (e.g., knowledge representation for bioinformatics discussed in [7]).

From a knowledge representation point of view, ontologies usually contain four components: (i) concepts that represent sets or classes of entities in a domain, (ii) instances that represent the actual entities, (iii) relations, and (iv) axioms that represent facts that are always true in the topic area of the ontology. Relations represent relation-

[3] All the material related to the prototype implementation (OBG-gen) is available online at https://github.com/LiUSemWeb/OBG-gen.

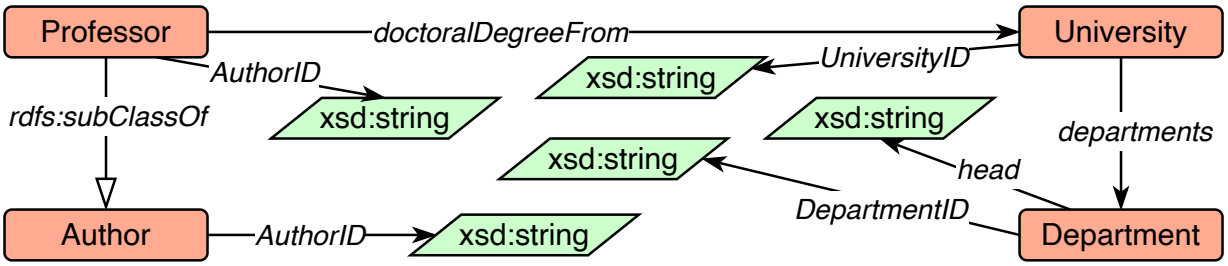


Figure 1. Example of an ontology.

ships among concepts. Axioms represent domain restrictions, cardinality restrictions, or disjointness restrictions. Depending on the components and information related to the components they contain, ontologies can be classified. Figure 1 represents an example ontology for the university domain. The open-headed arrows represent axioms that represent is-a relationships that is, if A is a B, then all entities belonging to concept A also belong to concept B. We say that A is a sub-concept of B. In this example `Professor` is a sub-concept of `Author`. Therefore, all `Professor` entities are `Author` entities. The closed-headed arrows represent general relations among concepts other than is-a relations. For instance, the `Professor` concept has a connection to the `University` concept represented by the `doctoralDegreeFrom` relation. Additionally, a relation can exist between a concept and a data type reference. For instance, `University` has a connection to the data type reference `xsd:string` represented by the `UniversityID` relation. This means that each entity of the `University` concept can be associated with a string type value by having a `UniversityID` connection.

To formally define the above concepts and relationships, we need representation languages. Description logics (DL) are a family of knowledge representation languages. There are three basic building blocks of such a language, namely: (i) atomic concepts (unary predicates) such as `University` and `Department`, (ii) atomic roles (binary predicates) such as `departments`, and (iii) individuals (constants) [8]. Complex concepts and roles can be built by using atomic concepts and logical constructors (e.g., conjunction ($\sqcap$), disjunction ($\sqcup$), universal restriction ($\forall$) and existential restriction ($\exists$)). Axioms can be defined using general concept inclusions ($\sqsubseteq$). For instance, the general concept inclusion (GCI) $\texttt{University} \sqsubseteq \forall \texttt{departments.Department}$ represents the fact that `University` is a sub-concept of the concept $\forall$`departments.Department` that represents all entities that may have `departments` relations to entities and if so, these latter entities must belong to the `Department` concept; an assertion `University`(Linköping University) means that the instance Linköping University belongs to the `University` concept. A DL TBox is a finite set of GCIs and a DL ABox is a finite set of assertions.

2.2. Data Integration

Data integration deals with combining data that resides at multiple different sources [9–11]. Ideally, a data integration system should enable unified access to a number of data sources [9, 11]. Formally, according to [9], a data integration system can be formalized as a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where,

- $\mathcal{G}$ is the *global schema*, expressed in a language $\mathcal{L}_\mathcal{G}$ over an alphabet $\mathcal{A}_\mathcal{G}$;
- $\mathcal{S}$ is the *source schema*, expressed in a language $\mathcal{L}_\mathcal{S}$ over an alphabet $\mathcal{A}_\mathcal{S}$;
- $\mathcal{M}$ is the *mapping* between $\mathcal{G}$ and $\mathcal{S}$, constituted by a set of *assertions* that define mappings from queries over the source schema $\mathcal{S}$ to queries over the global schema $\mathcal{G}$ (similarly for mappings from queries over $\mathcal{G}$ to queries over $\mathcal{S}$). Such a mapping specifies correspondences between concepts in the global schema and those in the source schema.

Ontology-based data integration (OBDI) is a form of data integration in which an ontology plays the role of a global schema that captures domain knowledge [12]. Usually, in an information system with only one single data source, the formal treatment of OBDI is identical to that of ontology-based data access (OBDA) [12, 13]. In this paper, we generally refer to both OBDI and OBDA as OBDA. OBDA, as a semantic technology, aims to facilitate access to different underlying data sources [14]. Traditionally, these underlying data sources are considered to be

relational databases. Ontologies play the role of global views over multiple data sources. There are different ways to implement an OBDA system. Generally, these systems can be categorized into two types, namely, data warehouse-based approaches and virtual approaches. These two categories of methods both make use of semantic mappings in order to overcome the differences between ontologies and local schemas, but in different ways [15, 16]. In a data warehouse-based approach, data from multiple sources are usually loaded or stored in a centralized storage, which is the *warehouse* [11, 17], based on semantic mappings. We refer to the data in such warehouses as materialized data. Depending on the aims or functionalities of a system, the materialized data could be stored in local databases or transformed into RDF graphs. Therefore, queries are evaluated against the materialized data. In a virtual approach, data is retained at the original sources and *mediators* are used to translate queries defined in terms of a global or mediated schema into queries defined in terms of each data source's local schema, based on semantic mappings. Therefore, queries are evaluated and executed against each data source. SPARQL queries are widely supported by data integration systems that use ontologies as global schemas.

A number of semantic mapping definition languages have been proposed over the years. One such language is R2RML (RDB to RDF Mapping Language), one of the two recommendations by the RDB2RDF W3C Working Group[4] to define semantic mappings [18]. It supports transformation rules defined by users, while the other recommendation, Direct Mapping [19], does not. Another language is RDF Mapping Language (RML) [20, 21], which allows underlying data in formats beyond relational databases and is a superset of R2RML. RML allows data from CSV, JSON, and XML data sources. In our work we make use of RML and we introduce RML in more details in Section 5.2.

2.3. GraphQL

GraphQL schemas and GraphQL resolver functions are basic building blocks in the implementations of GraphQL servers. The former describe how users can retrieve data using GraphQL APIs. The latter contain program code including how to access data sources and structure the obtained data according to the schema. We introduce GraphQL schemas and GraphQL resolver functions in Section 2.3.1 and Section 2.3.2, respectively.

2.3.1. GraphQL Schemas

In a GraphQL API, the GraphQL schema defines types, their fields, and the value types of the fields. Such a schema represents a form of vocabulary supported by a GraphQL API rather than specifying what the data instances of an underlying data source may look like and what constraints have to be guaranteed [22]. There are six different (named) type definitions in GraphQL, which are scalar type, object type, interface type, union type, enum type and input object type. Listing 1 depicts a GraphQL schema example.

An object type represents a list of fields and each field has a value of a specific type such as object type or scalar type. A scalar is used to represent a value such as a string. In Listing 1, there are three basic object type definitions, which are `University` (line 1 to line 4), `Department` (line 5 to line 8), and `Professor` (line 12 to line 15). They all have field definitions which represent the relationships to scalar types or to other object types. For instance, the `University` type has a field definition `UniversityID` of which the value type is `String` (line 2), and a field definition `departments` of which the value type is a list of `Departments` (line 3). GraphQL allows defining abstract types by supporting the interface type and the union type. An interface type defines a list of fields and allows object types to implement. An object type can then implement an interface type with the requirement that the object type includes all fields defined by the interface type. The schema in Listing 1 contains an interface type, `Author` with an `AuthorID` field of which the value type is `String` (line 9 to line 11). The object type `Professor` implements `Author` and must have the same definition for `AuthorID` field (line 13) as that in `Author`. A union type defines a list of possible types. An enum type describes the set of possible values that are in scalars. For more details of union and enum types, we refer the reader to the latest GraphQL specification in [1].

GraphQL allows fields to accept arguments to configure their behavior [1]. These arguments can be defined by input object types. An input object type defines an input object with a set of input fields; the input fields are either scalars, enums, or other input objects. This allows arguments to accept arbitrarily com-

[4] https://www.w3.org/2001/sw/rdb2rdf/

Listing 1: Example of a GraphQL schema.

```
1  type University{
2    UniversityID: String
3    departments: [Department]
4  }
5  type Department{
6    DepartmentID: String
7    head: String
8  }
9  interface Author{
10   AuthorID: String
11 }
12 type Professor implements Author{
13   AuthorID: String
14   doctoralDegreeFrom: [University]
15 }
16 input UniversityFilter{
17   UniversityID: StringFilter
18   departments: DepartmentFilter
19   _and: [UniversityFilter]
20   _or: [UniversityFilter]
21   _not: UniversityFilter
22 }
23 input DepartmentFilter{
24   DepartmentID: StringFilter
25   head: StringFilter
26   _and: [DepartmentFilter]
27   _or: [DepartmentFilter]
28   _not: DepartmentFilter
29 }
30 input StringFilter{
31   _eq: String
32   _in: [String]
33   _neq: String
34   _nin: [String]
35   _like: String
36 }
37 type Query{
38   UniversityList(filter: UniversityFilter): [University]
39   DepartmentList(filter: DepartmentFilter): [Department]
40   AuthorList: [Author]
41   ProfessorList: [Professor]
42 }
```

plex structs, which can capture notions of filtering conditions. For instance, according to the definitions of `UniversityFilter` (line 16 to line 22) and `StringFilter` (line 30 to line 36), we can define an input argument as `UniversityID:{_eq:"u1"}` to capture the meaning of "*UniversityID is equal to 'u1'*", where `_eq` represents the equal to operator. In our approach, `_and`, `_or` and `_not` are used to represent boolean expressions. For instance, `_or:[{UniversityID:{_eq:"u1"}},{UniversityID:{_eq:"u2"}}]` represents the expression "*UniversityID is equal to 'u1' or 'u2'*". In the example schema, we use the term `filter` to represent the name of an input argument (e.g., line 38). Such input arguments defined as input objects are not built-in constructs of GraphQL. Therefore, their meanings are essentially defined by the program code of the GraphQL server implementation, i.e., the resolver functions which manage requests to underlying data sources and structure the returned data according to the GraphQL schema. Our approach presented in Section 4 and Section 5 uses input arguments named as `filter` to represent filter conditions.

Listing 2: Example of a resolver function for the `UniversityList` field.

```
1 const UniversityList = (university_id) => {
2   let query = "";
3   if(university_id)
4     query = db_conection.select().from('university').where('id', university_id);
5   else
6     query = db_conection.select().from('university');
7   return query.then(rows => rows.map(row => new University(row)));
8 };
```

Additionally, a GraphQL schema supports defining types that represent operations such as query and mutation. The schema presumes the `Query` type as the query root operation type. As Listing 1 shows, in the `Query` type definition (line 37 to line 42), there are four field definitions, which are `UniversityList`, `DepartmentList`, `AuthorList`, and `ProfessorList`. For instance, the returned type of `UniversityList` is `[University]`, a list of `Universities`. The `UniversityList` takes an argument defined as `UniversityFilter` as an input for capturing the notion of a filtering condition.

2.3.2. *GraphQL Resolver Functions*

In a GraphQL API, apart from the GraphQL schema defining types, their fields, and the value types of the fields, resolver functions are responsible for populating the data for fields of types in the GraphQL schema. For instance, for the schema example shown in Listing 1, there are four fields defined in the `Query` type. Therefore, in the GraphQL server implementation, we are supposed to define resolver functions to populate data for these fields, `UniversityList`, `DepartmentList`, `AuthorList`, and `ProfessorList`. In our approach, we assume that the GraphQL schema supports a query that retrieves all the instances for each interface type or object type. Therefore, we use the name of each interface type or object type concatenated with 'List' as the name of a field in the `Query` type, where the returned type is a list of the interface or object type. This is a way to state the behavior of a field in the `Query` type. We emphasize that what a GraphQL query can retrieve over the underlying data sources relies on how the resolver function is implemented. For instance, if the underlying data source is a relational database, the resolver function should contain code specifying the SQL query to be evaluated. Listing 2 illustrates an example resolver function (written in JavaScript syntax) for the `UniversityList` field. We assume that the underlying data source is a relational database that contains a table named *university* with a column named *id*. In line 4, given an input argument (*university_id*) representing the id of a university, a query is evaluated against the relational database. In line 7, the data is structured according to the *University* object defined in the JavaScript code which corresponds to the `University` type definition in the schema shown in Listing 1.

3. GraphQL-Based Framework for Data Access and Data Integration

This section introduces an overview of the GraphQL-based framework for data access and integration and two basic processes in this framework.

3.1. *Overview of the Framework*

Figure 2 illustrates the framework for data access and integration based on GraphQL in which an ontology drives the generation of GraphQL server that provides integrated access to data from heterogeneous data sources. These data sources may be based on different schemas and formats and may be accessed in different ways (e.g., as tabular data accessed via SQL queries or as JSON-formatted data accessed via API requests). To address the heterogeneity, the framework relies on an ontology that provides an integrated view of the data from the different sources, and corresponding semantic mappings that define how the data from the underlying data sources is interpreted or annotated by the ontology (arrows **(a)**) and **(b)**). Furthermore, two processes are defined. The first process generates the GraphQL server. The second process deals with answering queries and is performed after the GraphQL server is set

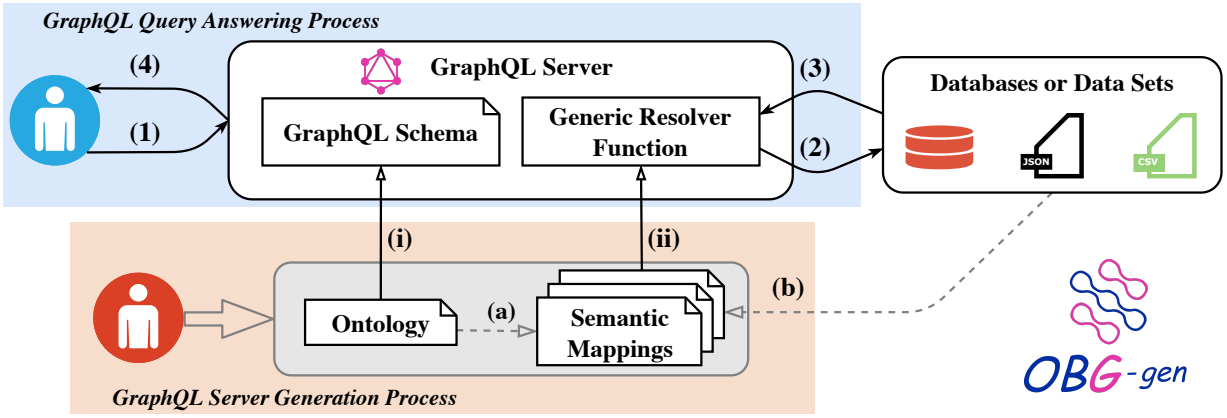


Figure 2. GraphQL-based framework for data access and integration.

up. In accordance with these two processes, there are two types of intended users or developers in the framework. One type is users or developers of the GraphQL server generator, who should have prior knowledge of the ontology, semantic mappings and the domain. The other type is end users using a GraphQL server for data access and integration, who may or may not be familiar with the Semantic Web or ontologies. For the purpose of writing GraphQL queries, they need basic prior knowledge of GraphQL, which can be learned from the self-documenting API of the generated GraphQL server showing the schema. We introduce more details about these two processes in Section 3.2 and Section 3.3, respectively.

3.2. GraphQL Server Generation Process

This process includes generating both a GraphQL schema for the API provided by the server (arrow **(i)**) and a generic resolver function (arrow **(ii)**). Given an ontology as an integrated view of data from multiple data sources, we propose a method for generating a GraphQL schema based on this ontology, with the result that the schema becomes a view of the data to be integrated. Additionally, we propose a generic implementation of resolver functions that takes semantic mappings as inputs, so that the server is able to get data from underlying data sources. In Sections 4 and 5, we elaborate on the implementation of our approaches for generating a GraphQL schema and the generic resolver function, respectively. This GraphQL server generation process does not need to be repeated unless the ontology or the semantic mappings change. After this generation process, the GraphQL server can be set up.

This process requires users or developers who are familiar with the query mechanisms of underlying data sources, domain ontologies that can be used for data access or integration. Consequently, they can define the scope of the ontology that will be used for generating the GraphQL schema for the server, as well as the semantic mappings that will be used for generating the generic resolver function. This type of automatic generation of GraphQL servers based on ontologies and semantic mappings can also benefit general GraphQL application developers, since it can eliminate the need to build GraphQL servers from scratch.

3.3. GraphQL Query Answering Process

During this process the query is validated against the GraphQL schema (arrow **(1)**); the underlying data sources are accessed via resolver functions, the retrieved data is combined, the data is structured according to the schema (arrows **(2)** and **(3)**); and finally the query result is returned (arrow **(4)**).

A GraphQL query example and corresponding query result are shown in Figure 3. The example query is: "*Get the university including the head of each department where the UniversityID is 'u1'*". The query takes as an input an argument defined as `filter:{UniversityID:{_eq:"u1"}}`, which follows the syntax of the input object type `UniversityFilter`. As mentioned in Section 2.3.1, the meaning of an input argument defined as an input object type is essentially determined by the program code of the resolver functions. Thus the query example shown

```
{
  UniversityList(
    filter:{
      UniversityID:{_eq:"u1"}
    })
  {
    departments{
      head
    }
  }
}
```

(a) Query.

```
{
  "data":{
    "UniversityList":
    [{
      "departments":[
        {"head":"Harry,Potter"},
        {"head":"Sheldon,Cooper"}
      ]
    }]
  }
}
```

(b) Query Response.

Figure 3. Example GraphQL query/response.

in Figure 3a illustrates one way that we make use of input objects to represent filtering conditions. In general, however, the input object types can be used in various ways for any field, depending on the implementation of the GraphQL server.

As mentioned earlier, domain users are the intended users of GraphQL servers, regardless of whether they have prior knowledge of the Semantic Web or ontologies. In order to write GraphQL queries, they only need to have a basic understanding of GraphQL, which can easily be explored via the GraphQL API provided by the server.

4. Ontology-Based GraphQL Schema Generation

As mentioned in Section 2.3.1, the GraphQL schema represents a form of vocabulary supported by the GraphQL API rather than specifying what the data instances of an underlying data source may look like and what constraints have to be guaranteed. Therefore, we focus on GraphQL language features supporting semantics-aware and integrated data access, namely how data can be queried, rather than reflecting the semantics of a complex knowledge representation language in the context of a GraphQL schema. Section 4.1 introduces how a GraphQL schema is formalized, and Section 4.2 introduces how an ontology is represented via a description logic TBox. Given an ontology represented in a description logic TBox, the concept and role names can be used to generate types and fields in a GraphQL schema. The general concept inclusions in a description logic TBox can be used to specify how to connect generated types and fields in a GraphQL schema. Then, in Section 4.3, we present the core algorithm (*Schema Generator*) for generating a GraphQL schema based on an ontology. In Section 4.4, we present the intended meaning of GraphQL schemas generated by the *Schema Generator*.

4.1. GraphQL Schema Formalization

According to [22, 23], a GraphQL schema can be defined over five finite sets. These five sets are $\mathtt{F} \subset \mathtt{Fields}$, $\mathtt{A} \subset \mathtt{Arguments}$, $\mathtt{T} \subset \mathtt{Types}$, $\mathtt{S} \subset \mathtt{Scalars}$, and $\mathtt{D} \subset \mathtt{Directives}$ where $\mathtt{T}$ is the disjoint union of $\mathtt{O}_\mathtt{T}$ (object types), $\mathtt{I}_\mathtt{T}$ (interface types), $\mathtt{U}_\mathtt{T}$ (union types), $\mathtt{IO}_\mathtt{T}$ (input object types) and $\mathtt{S}$ (scalar types). $\mathtt{Fields}$, $\mathtt{Arguments}$, $\mathtt{Types}$, and $\mathtt{Directives}$ are pairwise disjoint, countably infinite sets representing field names, argument names, type names, and directive names, respectively. $\mathtt{Scalars}$, which is a subset of $\mathtt{Types}$, represents five built-in scalar types, which are $\mathtt{Int}$, $\mathtt{Float}$, $\mathtt{String}$, $\mathtt{Boolean}$, and $\mathtt{ID}$. Moreover, the GraphQL schema definition language introduces *non-null types* and *list types*, called *wrapping types*, according to types in $\mathtt{Types}$. Given a type $\mathtt{t}$ belonging to $\mathtt{Types}$, the former is denoted as $\mathtt{t!}$, while the latter is denoted as $[\mathtt{t}]$. $\mathtt{W}_\mathtt{T}$ is used to denote the set of all types that can be formed by wrapping the types in $\mathtt{T}$, and $\mathtt{W}_\mathtt{S}$ denotes the set of all types that can be formed by wrapping the scalar types in $\mathtt{S}$. In our current work, considering the knowledge representation language we use for the ontology (see next section), we do not need directive and union types. Therefore, a GraphQL schema $\mathcal{S}$ is defined over $(\mathtt{F}, \mathtt{A}, \mathtt{T}, \mathtt{S})$ consisting of two assignments that are $type_\mathcal{S}$ and $implementation_\mathcal{S}$:

- $type_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup type^{\mathrm{AF}}_{\mathcal{S}}$ where,
  - $type^{\mathrm{F}}_{\mathcal{S}} : (\mathtt{O_T} \cup \mathtt{I_T} \cup \mathtt{IO_T}) \times \mathtt{F} \rightharpoonup \mathtt{T} \cup \mathtt{W_T}$, which is a partial function since a type has a set of fields which is a subset of $\mathtt{F}$, assigns a type to each field that is defined for an object type, an interface type or an input object type,
  - $type^{\mathrm{AF}}_{\mathcal{S}} : dom(type^{\mathrm{F}}_{\mathcal{S}}) \times \mathtt{A} \rightharpoonup \mathtt{S} \cup \mathtt{W_S} \cup \mathtt{IO_T}$, which is a partial function since a field has a set of arguments which is a subset of $\mathtt{A}$, assigns a type to every argument of fields that are defined for a type;
- $implementation_{\mathcal{S}} : \mathtt{I_T} \to 2^{\mathtt{O_T} \cup \mathtt{I_T}}$ assigns a set of object types or interface types to every interface type.

Figure 4 illustrates a formalized representation of the GraphQL schema shown in Listing 1. In the formalization, we have sets $\mathtt{F}$, $\mathtt{A}$, $\mathtt{I_T}$, $\mathtt{O_T}$, $\mathtt{S}$ and $\mathtt{IO_T}$, which contains all the field names, argument names, interface type names, object type names, scalar type names and input object type names, respectively. Additionally, the formal-

- $\mathtt{F} = \{$UniversityID, departments, DepartmentID, head, AuthorID, doctoralDegreeFrom, _and, _or, _not, _eq, _in, _neq, _nin, _like, UniversityList, DepartmentList, AuthorList, ProfessorList$\}$;
- $\mathtt{A} = \{$filter$\}$;
- $\mathtt{T} = \mathtt{I_T} \cup \mathtt{O_T} \cup \mathtt{S} \cup \mathtt{IO_T}$ where,
  - $\mathtt{I_T} = \{$Author$\}$,
  - $\mathtt{O_T} = \{$Query, University, Department, Professor$\}$,
  - $\mathtt{S} = \{$String$\}$,
  - $\mathtt{IO_T} = \{$UniversityFilter, DepartmentFilter, StringFilter$\}$;
- $type_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup type^{\mathrm{AF}}_{\mathcal{S}}$ where,
  - $type^{\mathrm{F}}_{\mathcal{S}} = \{$(University, UniversityID) $\mapsto$ String,
    (University, departments) $\mapsto$ [Department],
    (Department, DepartmentID) $\mapsto$ String,
    (Department, head) $\mapsto$ String,
    (Author, AuthorID) $\mapsto$ String,
    (Professor, AuthorID) $\mapsto$ String,
    (Professor, doctoralDegreeFrom) $\mapsto$ [University],
    (UniversityFilter, UniversityID) $\mapsto$ StringFilter,
    (UniversityFilter, departments) $\mapsto$ DepartmentFilter,
    (UniversityFilter, _and) $\mapsto$ [UniversityFilter],
    (UniversityFilter, _or) $\mapsto$ [UniversityFilter],
    (UniversityFilter, _not) $\mapsto$ UniversityFilter,
    (DepartmentFilter, DepartmentID) $\mapsto$ StringFilter,
    (DepartmentFilter, head) $\mapsto$ StringFilter,
    (DepartmentFilter, _and) $\mapsto$ [DepartmentFilter],
    (DepartmentFilter, _or) $\mapsto$ [DepartmentFilter],
    (DepartmentFilter, _not) $\mapsto$ DepartmentFilter,
    (StringFilter, _eq) $\mapsto$ String,
    (StringFilter, _in) $\mapsto$ [String],
    (StringFilter, _neq) $\mapsto$ String,
    (StringFilter, _nin) $\mapsto$ [String],
    (StringFilter, _like) $\mapsto$ String,
    (Query, UniversityList) $\mapsto$ [University],
    (Query, DepartmentList) $\mapsto$ [Department],
    (Query, AuthorList) $\mapsto$ [Author],
    (Query, ProfessorList) $\mapsto$ [Professor]$\}$;
  - $type^{\mathrm{AF}}_{\mathcal{S}} = \{$((Query, UniversityList), filter) $\mapsto$ UniversityFilter,
    ((Query, DepartmentList), filter) $\mapsto$ DepartmentFilter$\}$;
- $implementation_{\mathcal{S}} = \{$Author $\mapsto$ {Professor}$\}$.

Figure 4. The formalization of the GraphQL schema shown in Listing 1.

Table 1
The syntax and semantics for the description logic used in our approach.

	Name	Syntax	Semantics
Construct	Top concept	$\top$	$\Delta^{\mathcal{I}}$
	Atomic concept	P	$P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
	Role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
	Attribute	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$
	Datatype	d	$d^{\mathcal{I}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$
	Conjunction	$P \sqcap Q$	$P^{\mathcal{I}} \cap Q^{\mathcal{I}}$
	Role value restriction	$\forall R.P$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta^{\mathcal{I}} : (x,y) \in R^{\mathcal{I}} \rightarrow y \in P^{\mathcal{I}}\}$
	Role qualified number restriction	$= 1R.P$	$\{x \in \Delta^{\mathcal{I}} \mid \lvert\{y \in \Delta^{\mathcal{I}} \mid (x,y) \in R^{\mathcal{I}} \wedge y \in P^{\mathcal{I}}\}\rvert = 1\}$
	Attribute value restriction	$\forall A.d$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta_{\mathbf{D}}^{\mathcal{I}} : (x,y) \in A^{\mathcal{I}} \rightarrow y \in d^{\mathcal{I}}\}$
	Attribute qualified number restriction	$= 1A.d$	$\{x \in \Delta^{\mathcal{I}} \mid \lvert\{y \in \Delta_{\mathbf{D}}^{\mathcal{I}} \mid (x,y) \in A^{\mathcal{I}} \wedge y \in d^{\mathcal{I}}\}\rvert = 1\}$
TBox	GCI	$P \sqsubseteq Q$	$P^{\mathcal{I}} \subseteq Q^{\mathcal{I}}$
ABox	Concept assertion	$P(a)$	$a^{\mathcal{I}} \in P^{\mathcal{I}}$
	Role assertion	$R(a,b)$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$
	Attribute assertion	$A(a,v)$	$(a^{\mathcal{I}}, v^{\mathcal{I}}) \in A^{\mathcal{I}}$

ization contains field declarations in the set $type_{\mathcal{S}}^{\mathtt{F}}$; argument declarations in $type_{\mathcal{S}}^{\mathtt{AF}}$; object types implementing interface types declarations in $implementation_{\mathcal{S}}$. For instance, $(\mathtt{University}, \mathtt{UniversityID}) \mapsto \mathtt{String}$ declares that the `University` type has a field `UniversityID` of which the returned type is `String`; $((\mathtt{Query}, \mathtt{UniversityList}), \mathtt{filter}) \mapsto \mathtt{UniversityFilter}$ declares that the `UniversityList` field accepts an input argument which is defined as the type `UniversityFilter`; $\mathtt{Author} \mapsto \{\mathtt{Professor}\}$ declares that the `Professor` type is one of the types that implement the interface `Author`.

4.2. Ontology Representation by a Description Logic TBox

In this work we assume that the ontology is represented by a TBox in a description logic which is an extension of $\mathcal{FL}_0$ by adding qualified number restrictions and datatypes. $\mathcal{FL}_0$ allows atomic concepts, the universal concept, intersection and value restriction [24]. This description logic can represent the semantics that can be reflected in a GraphQL schema for data access and integration where the schema follows the GraphQL schema definition language. Note that we do not aim to represent the full ontology in DL, but that, for our purposes, we only need the part of the ontology that is needed for data access. Therefore, we only use the DL constructors that cover the existing GraphQL features for data access.

The syntax and semantics of the description logic used in our approach are shown in Table 1. The introduction of datatypes is based on the work presented in [25] and [26]. Let $\mathsf{N_C}$, $\mathsf{N_R}$, $\mathsf{N_A}$, and $\mathbf{D}$ be disjoint finite sets of concept names, role names, attribute names, and datatype names respectively. An interpretation $\mathcal{I}$ consists of a non-empty set $\Delta^{\mathcal{I}}$ representing the domain of individuals, and an interpretation function $\cdot^{\mathcal{I}}$. In addition, the interpretation includes an interpretation domain for data values $\Delta_{\mathbf{D}}^{\mathcal{I}}$ which is disjoint from the domain of individuals $\Delta^{\mathcal{I}}$ [25]. A datatype, such as integer, is interpreted as a subset of $\Delta_{\mathbf{D}}^{\mathcal{I}}$ and a value such as the integer 5 is interpreted as an element of $\Delta_{\mathbf{D}}^{\mathcal{I}}$. Thus, the interpretation function $\cdot^{\mathcal{I}}$ assigns to each atomic concept $P \in \mathsf{N_C}$ a subset $P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, to each $d \in \mathbf{D}$ a set $d^{\mathcal{I}} \subseteq \Delta_{\mathbf{D}}^{\mathcal{I}}$, to each role $r \in \mathsf{N_R}$ a relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, to each attribute $a \in \mathsf{N_A}$ a relation $a^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta_{\mathbf{D}}^{\mathcal{I}}$, to each individual name i an element $i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, and to each data value v an element $v^{\mathcal{I}} \in \Delta_{\mathbf{D}}^{\mathcal{I}}$.

A TBox over $\mathsf{N_C}$, $\mathsf{N_R}$, $\mathsf{N_A}$ and $\mathbf{D}$ is a finite set of general concept inclusions (GCI) where each GCI is a statement in the form of $B \sqsubseteq C$, where B and C are concepts. For generating GraphQL schemas, we use normalized TBoxes that contain only GCIs in the normal forms given in formula 1 where $P, Q \in \mathsf{N_C}$, $r \in \mathsf{N_R}$, $a \in \mathsf{N_A}$, and $d \in \mathbf{D}$. There

$$NF_1 : P \sqsubseteq Q \qquad NF_2 : P \sqsubseteq \forall r.Q \qquad NF_3 : P \sqsubseteq = 1r.Q \qquad NF_4 : P \sqsubseteq \forall a.d \qquad NF_5 : P \sqsubseteq = 1a.d \tag{1}$$

$\mathsf{N_C}$ = {University, Department, Author, Professor}, $\mathsf{N_R}$ = {departments, doctoralDegreeFrom},
$\mathsf{N_A}$ = {UniversityID, DepartmentID, head, AuthorID}, $\mathbf{D}$ = {xsd:string},
University $\sqsubseteq \forall$ departments.Department
University $\sqsubseteq =1$ UniversityID.xsd:string
Department $\sqsubseteq =1$ DepartmentID.xsd:string
Department $\sqsubseteq =1$ head.xsd:string
Author $\sqsubseteq =1$ AuthorID.xsd:string
Professor $\sqsubseteq$ Author
Professor $\sqsubseteq =1$ AuthorID.xsd:string
Professor $\sqsubseteq \forall$ doctoralDegreeFrom.University

Figure 5. Example of a TBox.

exist normalization rules to obtain such a TBox (e.g., an axiom $P \sqsubseteq Q \sqcap M$ is converted to two axioms, $P \sqsubseteq Q$ and $P \sqsubseteq M$) [27]. Baader et al. show that such normalization rules can preserve a conservative extension of a TBox in $\mathcal{FL}_0$ [28]. A conservative extension guarantees that subsumptions according to the original TBox coincide with those with respect to the normalized TBox. An example TBox input is shown in Figure 5.

4.3. Schema Generator Algorithm

Algorithm 1 shows the details to generate a GraphQL schema. The output for the example is the schema shown in Listing 1. First, the algorithm iterates over the concept names in $\mathsf{N_C}$ (line 1 to line 5). For each concept, such as `University` in the TBox, the concept name (`University`) is used as the name of an object type to be generated (line 2); the term concatenated with 'Filter' is used as the name of an input type (`UniversityFilter`) to be generated (line 3); the term concatenated with 'List' is used as the name of a field (`UniversityList`) of the `Query` type (line 4). Additionally, each such field of the `Query` type is assigned an argument named 'filter', with a type that is the corresponding input type (e.g., `filter:UniversityFilter` to `UniversityList`) (line 5). Next, the algorithm iterates over GCIs in the TBox (line 6 to line 30). For a GCI in the form of NF_1, the name of the super-concept is used as the name of an interface type to be generated; a field for the `Query` type named by concatenating the interface type name and 'List' is generated; the previously generated object type corresponding to the sub-concept implements the generated interface type.

From line 13 to line 21, the algorithm deals with GCIs containing roles, which can be of the form NF_2 or NF_3 (such as `University` $\sqsubseteq \forall$ `departments.Department`). In both cases, a field definition (e.g., `departments`) of the object type (e.g., `University`) and a field definition (`departments`) of the input type (`UniversityFilter`) are generated. However, for NF_3, the returned type of the field is defined as the original object type corresponding to the concept appearing on the right side of the GCI (line 20). For NF_2, the returned type is defined as a wrapped type, which is a list type (line 17). For instance, the `departments` field declaration for the `University` type is `departments:[Department]`. The algorithm deals with GCIs containing attributes in a similar way (line 22 to line 30). For example, the `University` object type has a field declaration, which is `UniversityID:String`.

We define a function Φ for mapping a datatype that exists in the TBox to a scalar type in GraphQL. Due to the fact that current GraphQL supports five basic scalar types which are `ID`, `Float`, `Int`, `Boolean`, and `String`, our current implementation of function Φ focuses on mapping datatypes `xsd:float`, `xsd:int`, `xsd:string` and `xsd:boolean` to scalar types `Float`, `Int`, `String` and `Boolean`, respectively. However, GraphQL allows users to define custom scalar types, and the values of such custom types should be JSON serializable. Therefore, our Φ function can be extended in the future for mapping any datatype besides the above four types from a TBox into a custom scalar type in GraphQL.

By generating the GraphQL schema based on an ontology, the schema will contain object or interface types corresponding to concepts in the ontology, and field declarations corresponding to relationships in the ontology. When a GraphQL query is sent to the GraphQL server, a resolver function parses the query to determine which type in the schema is requested. It then parses the relevant definitions corresponding to such a type in the semantic mappings to

Algorithm 1: *Schema Generator*

Input : $\mathsf{N_C}$; normalized TBox $\mathcal{TB}$; Φ, mapping a datatype in **D** to a scalar type
Output: a GraphQL schema $\mathcal{S}$

1 **for** $P \in \mathsf{N_C}$ **do**
2 $\quad \mathtt{O_T} = \mathtt{O_T} \cup \{P\}$ `// extend` $\mathcal{S}$ `with an empty object type,` P
3 $\quad \mathtt{IO_T} = \mathtt{IO_T} \cup \{P\mathtt{Filter}\}$ `// extend` $\mathcal{S}$ `with an empty input type,` $P\mathtt{Filter}$
$\quad$ `/* add following field/argument declarations to the Query type */`
4 $\quad type^{\mathrm{F}}_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup \{(\mathtt{Query}, P\mathtt{List}) \mapsto [P]\}$
5 $\quad type^{\mathrm{AF}}_{\mathcal{S}} = type^{\mathrm{AF}}_{\mathcal{S}} \cup \{((\mathtt{Query}, P\mathtt{List}), \mathtt{filter}) \mapsto P\mathtt{Filter}\}$
6 **for** $t \in \mathcal{TB}$ **do**
7 $\quad$ **if** t is of the form $P \sqsubseteq Q$ (i.e., NF_1) **then**
8 $\quad\quad \mathtt{I_T} = \mathtt{I_T} \cup \{Q\}$ `// extend` $\mathcal{S}$ `with an empty interface type,` Q
9 $\quad\quad \mathtt{IO_T} = \mathtt{IO_T} \cup \{Q\mathtt{Filter}\}$ `// extend` $\mathcal{S}$ `with an input type,` $Q\mathtt{Filter}$
$\quad\quad$ `/* add following field/argument declarations to the Query type */`
10 $\quad\quad type^{\mathrm{F}}_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup \{(\mathtt{Query}, Q\mathtt{List}) \mapsto [Q]\}$
11 $\quad\quad type^{\mathrm{AF}}_{\mathcal{S}} = type^{\mathrm{AF}}_{\mathcal{S}} \cup \{((\mathtt{Query}, Q\mathtt{List}), \mathtt{filter}) \mapsto Q\mathtt{Filter}\}$
12 $\quad\quad implementation_{\mathcal{S}}(Q) = implementation_{\mathcal{S}}(Q) \cup P$ `// declare that the object type` P `implements` Q
13 $\quad$ **if** t is of the form of $P \sqsubseteq \forall r.Q$ (i.e., NF_2) **then**
14 $\quad\quad$ **if** $P \sqsubseteq= 1r.Q \in \mathcal{TB}$ **then**
15 $\quad\quad\quad$ Do nothing, this case will be handed in line 19 to line 21
16 $\quad\quad$ **else**
$\quad\quad\quad$ `/* add following field declarations to` P `and` $P\mathtt{Filter}$ `*/`
17 $\quad\quad\quad type^{\mathrm{F}}_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup \{(P, r) \mapsto [Q]\}$ `//` r`: [`Q`]`
18 $\quad\quad\quad type^{\mathrm{F}}_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup \{(P\mathtt{Filter}, r) \mapsto Q\mathtt{Filter}\}$ `//` r`:` $Q\mathtt{Filter}$
19 $\quad$ **if** t is of the form of $P \sqsubseteq= 1r.Q$ (i.e., NF_3) **then**
$\quad\quad$ `/* add following field declarations to` P `and` $P\mathtt{Filter}$ `*/`
20 $\quad\quad type^{\mathrm{F}}_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup \{(P, r) \mapsto Q\}$ `//` r`:` Q
21 $\quad\quad type^{\mathrm{F}}_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup \{(P\mathtt{Filter}, r) \mapsto Q\mathtt{Filter}\}$ `//` r`:` $Q\mathtt{Filter}$
22 $\quad$ **if** t is of the form of $P \sqsubseteq \forall a.d$ (i.e, NF_4) **then**
23 $\quad\quad$ **if** $P \sqsubseteq= 1a.d \in \mathcal{TB}$ **then**
24 $\quad\quad\quad$ Do nothing, this case will be handed in line 28 to line 30
25 $\quad\quad$ **else**
$\quad\quad\quad$ `/* add following field declarations to` P `and` $P\mathtt{Filter}$ `*/`
26 $\quad\quad\quad type^{\mathrm{F}}_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup \{(P, r) \mapsto [\Phi(d)]\}$ `//` r`: [`$\Phi(d)$`]`
27 $\quad\quad\quad type^{\mathrm{F}}_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup \{(P\mathtt{Filter}, r) \mapsto \Phi(d)\mathtt{Filter}\}$ `//` r`:` $\Phi(d)\mathtt{Filter}$
28 $\quad$ **if** t is of the form of $P \sqsubseteq= 1a.d$ (i.e., NF_5) **then**
$\quad\quad$ `/* add following field declarations to` P `and` $P\mathtt{Filter}$ `*/`
29 $\quad\quad type^{\mathrm{F}}_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup \{(P, r) \mapsto \Phi(d)\}$ `//` r`:` $\Phi(d)$
30 $\quad\quad type^{\mathrm{F}}_{\mathcal{S}} = type^{\mathrm{F}}_{\mathcal{S}} \cup \{(P\mathtt{Filter}, r) \mapsto \Phi(d)\mathtt{Filter}\}$ `//` r`:` $\Phi(d)\mathtt{Filter}$

retrieve data. For instance, if a query requests all the entities of the `University` object type, the resolver function parses the semantic mappings that are defined for the *University* concept to get information regarding how to access the underlying data sources.

4.4. The Intended Meaning of a Generated GraphQL Schema

In Section 4.3, we present the *Schema Generator* which takes a TBox representing an ontology as an input, to generate a GraphQL schema. Such a GraphQL schema can describe how to access underlying data sources in which the data can be annotated by the ontology. The underlying data can thus be viewed as an ABox for the TBox. Therefore, a GraphQL query that conforms to this GraphQL schema can be considered as a query over the ABox. To make this intention more formal we consider an ABox $\mathcal{A}$ as a finite set of assertions of the form $P(x)$, $R(x,y)$ or

University(university_1), University(university_2);
Department(d1), Department(d2), Department(d3), Department(d4);
departments(university_1, d1), departments(university_1, d2),
departments(university_2, d3), departments(university_2, d4);
UniversityID(university_1, "u1"), UniversityID(university_2, "u2");
head(d1, "Harry, Potter"), head(d2, "Sheldon, Cooper"),
head(d3, "Paul, Atreides"), head(d4, "Jack, Lee").

Figure 6. Example of an ABox.

$A(x, z)$, where $P \in \mathsf{N_c}, R \in \mathsf{N_R}, A \in \mathsf{N_A}$, x and y are instance names, z are literals. Figure 6 illustrates an example ABox based on the TBox in Figure 5.

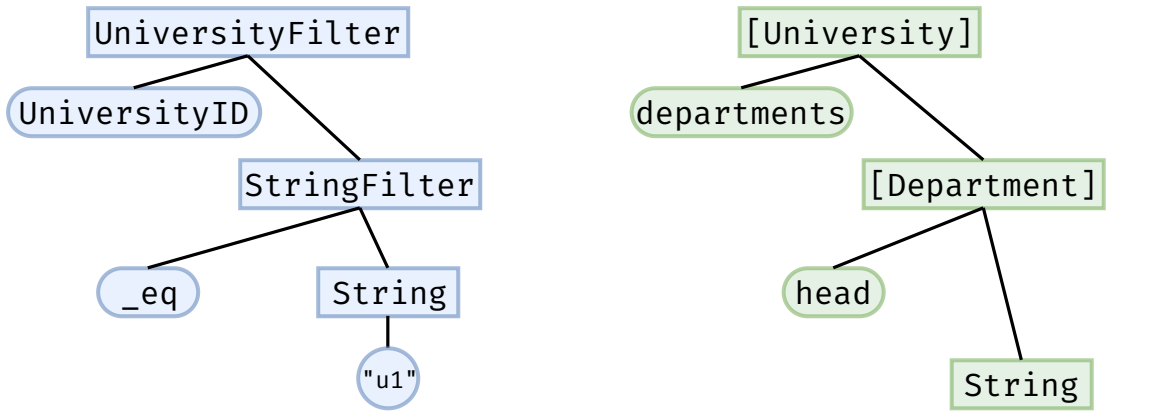
Let $\mathcal{O}$ be an ontology represented by a TBox $\mathcal{T}$; let $\mathcal{S}$ be a GraphQL schema over ($\mathcal{F}$, $\mathcal{A}$, $\mathcal{T}$, $\mathcal{S}$) generated by Algorithm 1 based on $\mathcal{T}$; let $\mathcal{Q}$ be a GraphQL query over ($\mathcal{F}$, $\mathcal{A}$, $\mathcal{T}$, $\mathcal{S}$) such that $\mathcal{Q}$ conforms to $\mathcal{S}$. Evaluating $\mathcal{Q}$ over underlying data sources that are instantiated in terms of $\mathcal{O}$ can be defined as retrieving an ABox $\mathcal{A}$ based on $\mathcal{T}$:

- If $\mathcal{Q}$ requests an object or an interface type t with a field f of which the returned type is a scalar type s or the wrapping type $[s]$ (i.e., $t \in \mathtt{O_T} \sqcup \mathtt{I_T}$, $f \in \mathtt{F}$, $s \in \mathtt{S}$, and $(t, f) \mapsto s \in type^{\mathtt{F}}_{\mathcal{S}}$ or $(t, f) \mapsto [s] \in type^{\mathtt{F}}_{\mathcal{S}}$), we can find the corresponding assertions in the ABox $\mathcal{A}$ of forms: $t(x)$ and $f(x, z)$;
- If $\mathcal{Q}$ requests an object or an interface type t_1 with a field f of which the returned type is another object or interface type t_2 or the wrapping type $[t_2]$ (i.e., $t_1, t_2 \in \mathtt{O_T} \sqcup \mathtt{I_T}$, $f \in \mathtt{F}$, and $(t_1, f) \mapsto t_2 \in type^{\mathtt{F}}_{\mathcal{S}}$ or $(t_1, f) \mapsto [t_2] \in type^{\mathtt{F}}_{\mathcal{S}}$), we can find the corresponding assertions in the ABox $\mathcal{A}$ of forms: $t_1(x)$, $t_2(y)$ and $f(x, y)$.

For instance, given the query (cf. Figure 3a) and the ABox (cf. Figure 6), University(university_1), departments(university_1, d1), departments(university_1, d2), head(d1, "Harry, Potter"), head(d2, "Sheldon, Cooper") are supposed to be retrieved. The above definition presents the meaning of the GraphQL schema generated based on a TBox for evaluating GraphQL queries. The definition relies on the *Schema Generator* where for each concept, the algorithm creates a corresponding type with the same name of the concept, same for roles and attributes. This guarantees to find the corresponding assertions from the ABox. However, in practice, as we presented in Section 2.3.2, how a GraphQL query retrieves data over the underlying data sources, depends on how the resolver function is implemented when we construct GraphQL servers. In the next section, we present how resolver functions can be implemented in a generic way based on semantic mappings.

5. Generic GraphQL Resolver Function

In general, there are two styles for implementing resolver functions for a GraphQL server. One option is to implement one resolver function per type (object or interface) defined in the GraphQL schema, where such a function states how to fetch the data to populate relevant fields. For instance, since the Query type in Listing 1 has four field definitions (UniversityList, DepartmentList, AuthorList, and ProfessorList), we may provide four resolver functions for getting entities of the University, Department, Author and Professor types from underlying data sources, respectively. The other option is to provide a resolver function for every field of every type defined in the GraphQL schema, such that this resolver could return data for this field of any type. In our framework, we adopt the first style because it can be easily generalized based on semantic mappings. That is, we implement a generic resolver function that can be used to populate objects of any object type or interface type, and can be viewed as a built-in function of the GraphQL server. In Section 5.1, we introduce how a GraphQL query is represented by Abstract Syntax Trees (ASTs), in which one represents query fields and others represent the filter expression. Section 5.2 introduces the RDF Mapping Language (RML), which is used for representing semantic mappings, and Section 5.3 describes the components of the generic resolver function. In Section 5.4, we present the core algorithm for the generic resolver function, which is responsible for accessing underlying data sources based on semantic mappings.



(a) Example Abstract Syntax Tree for filter fields.

(b) Example Abstract Syntax Tree for query fields.

Figure 7. Example Abstract Syntax Trees for the query shown in Figure 3a.

5.1. GraphQL Queries Represented by Abstract Syntax Trees

In general, a GraphQL query can be represented using a single AST that contains nodes representing the fields requested in the query, and also contains additional nodes for the input arguments that may be used for each of these fields. In our approach, we assume that each query accepts an input argument which captures the notion of a filter condition. Therefore, we specify the query evaluation in two steps: (i) evaluating for a filter condition, which is represented via an input argument that is defined as an input object type in the schema, (ii) evaluating for those fields that are requested in the GraphQL query. For instance, in the query example shown in Figure 3a, the field having a filtering condition is different from the requested fields (the former is `UniversityID` while the latter includes `departments` and `head`). In the evaluation step for the filter condition, the identifier information of the filtered out instances of the requested type (i.e., `University`) are obtained after accessing the underlying data sources. In the next step, the underlying data sources are accessed again to retrieve only the requested fields for the filtered instances. Therefore, to enable such two steps in the query evaluation, we use multiple ASTs to represent a GraphQL query (cf. Figure 7, these two ASTs represent the query shown in Figure 3a), one of which captures the input argument structure (Figure 7a), and the other of which captures the structure of the query, including the requested fields and their types (Figure 7b). More specifically, every node in such ASTs represents either a named type (i.e., object type, interface type, input type, or scalar type), a wrapping type, or a field. Additionally, ASTs that represent input arguments also contain nodes that represent the values of scalar-typed fields (e.g., `"u1"` in the AST shown in Figure 7a). The types (i.e., `UniversityFilter`, `StringFilter`, `String`) or wrapping types (i.e., `[University]`, `[Department]`) are drawn with rectangle nodes. The fields (i.e., `UniversityID`, `_eq`, `departments`, `head`) are drawn with rounded rectangle nodes.

In practice, a filter condition is converted into disjunctive normal form (DNF).[5] A query result in DNF contains data formed by the union of data that satisfies each disjunct. Therefore, in the step of evaluating for a filter condition: (i) multiple ASTs are generated where each represents one of the disjuncts, (ii) the underlying data source are accessed several times to obtain instances satisfying each disjunct, (iii) a union of identifier information for these instances of the requested type is returned.

5.2. RDF Mapping Language (RML)

RML is a declarative mapping language for linking data to ontologies [29]. An RML document has one or more `Triples Maps`, which declare how input data is mapped into triples of the form (subject, predicate, object).

[5] A statement is in DNF if it is a disjunction of conjunctions of literals. A disjunction uses the *OR* ($\vee$) operator. A conjunction uses the *AND* ($\wedge$) operator.

An example of RML mappings is shown in Listing 3. A `Triples Map` contains the following three components (`Logical Source`, `Subject Map` and a set of `Predicate-Object Maps`). A logical source declares the source of input data to be mapped. It contains definitions of `source` that locate the input data source, `reference formulation` declaring how to refer to the input data, and `logical iterator` declaring the iteration loop used to map the input data. For instance, line 2 to line 6 in Listing 3 constitute the definition of a logical source. The definition declares that the data source is a JSON-formatted data source on the Web and also describes the way of iterating the JSON-formatted data (line 5). A subject map declares a rule for generating subjects when transforming

Listing 3: Example of RML mappings transforming university domain data.

```
<UniversityMapping>
 rr:logicalSource [
   rml:source "http://example.com/universities.json";
   rml:referenceFormulation ql:JSONPath;
   rml:iterator "$.data.universities[*]";
 ];
 rr:subjectMap [
   rr:template "http://example.com/university/{uid}";
   rr:class schema:University;
 ];
 rr:predicateObjectMap [
   rr:predicate schema:UniversityID;
   rr:objectMap [
     rml:reference "uid";
   ];
 ];
 rr:predicateObjectMap [
   rr:predicate schema:departments;
   rr:objectMap [
     rr:parentTriplesMap <DepartmentMapping>
     rr:joinCondition [
      rr:child "uid";
      rr:parent "university_id";
     ];
   ];
 ].

<DepartmentMapping>
 rr:logicalSource [
   rml:source "http://example.com/departments.csv";
   rml:referenceFormulation ql:CSV;
 ];
 rr:subjectMap [
   rr:template "http://example.com/department/{department_id}";
   rr:class schema:Department;
 ];
 rr:predicateObjectMap [
   rr:predicate schema:DepartmentID;
   rr:objectMap [
     rml:reference "department_id";
   ];
 ];
 rr:predicateObjectMap [
   rr:predicate schema:head;
   rr:objectMap [
    rml:reference "HEAD";
   ];
 ].
```

underlying data into triples, including how to construct URIs of subjects (e.g., line 8) and specifying the concept to which subjects belong (e.g., line 9). A predicate-object map consists of one or more predicate maps declaring how to generate predicates of triples (e.g., line 12), and one or more object maps or referencing object maps defining how to generate objects of triples. An object map can be a reference-valued term map or a constant-valued term map. The former declares a valid reference to a column (relational data sources), or to an object (JSON data sources). The latter declares the value of the object as constant data. For instance, line 39 to line 41 make up a reference-valued term map. Line 19 to line 25 constitute a definition of a referencing object map including the join condition based on two triples maps. A referencing object map refers to another triples map (called a parent triples map) by using a `rr:joinCondition` property to state the join condition between the current triples map and the parent triples map. A join condition contains two properties, `rr:child` and `rr:parent`, of which the values must be logical references to logical sources of the current triples map and the parent triples map, respectively.

5.3. Components of the Generic Resolver Function

We show the basic technical components of the generic resolver function including *QueryParser* and *Evaluator* in Figure 8. In Algorithm 2, we show the generic resolver function. The inputs to the generic resolver function are a GraphQL schema, a GraphQL query and semantic mappings. The GraphQL query and schema are inputs of the *QueryParser*. The *QueryParser* parses a query including a filter expression given as an input argument, and outputs the corresponding ASTs (cf. Figure 7) for the input argument and the query structure, respectively (shown as arrows ① and ② in Figure 8). As mentioned in Section 5.1, in our practical solution a filter condition is converted into disjunctive normal form. In Algorithm 2, the *QueryParser* parses the query, converts a filter expression into a union of conjunctive expressions, and generates an AST for each conjunctive expression and an AST for the query structure (line 2). Then, the filter expression (line 5 to line 7 in Algorithm 2, frame ⓐ in Figure 8) and the query fields (line 9 and line 13 in Algorithm 2, frame ⓑ in Figure 8) are evaluated. The *Evaluator* is responsible for sending requests to underlying data sources and fetching data according to an AST. During evaluation of the filter expression, for each AST representing a conjunctive (sub-)expression, an evaluator is called to request data that satisfies the conjunctive (sub-)expression (line 6). After a call to an evaluator based on an AST (*filter_ast* in line 6), data representing the requested type, which contains identifier information, is returned (*identifier_info* in line 6). Taking the query in Figure 3a represented by the ASTs shown in Figure 7 as an example, the requested type is `University` and data that can identify university instances is supposed to be returned in *identifier_info*. Such identifier information is captured in semantic mappings, which are used to construct the URIs for subjects where such subjects represent instances of the *University* concept. For instance, in line 8 of the RML mappings example in Listing 3, the values of the *uid* attribute of the underlying data source are used to construct URIs of subjects representing instances of the *University* concept. The identifier information returned by evaluating each *filter_ast* is merged into *filtered_identifiers* (line 7). During evaluation of the query fields, such merged identifier information is taken into account in the call to the evaluator of the query fields (line 9 in Algorithm 2, arrow ③ in Figure 8).

As mentioned in Section 4.3, by generating the GraphQL schema based on an ontology, we can therefore, for each object or interface type and each field declaration, find the corresponding concept and relationship in the ontology. Since such concepts and relationships are used to define semantic mappings, when a generic resolver function

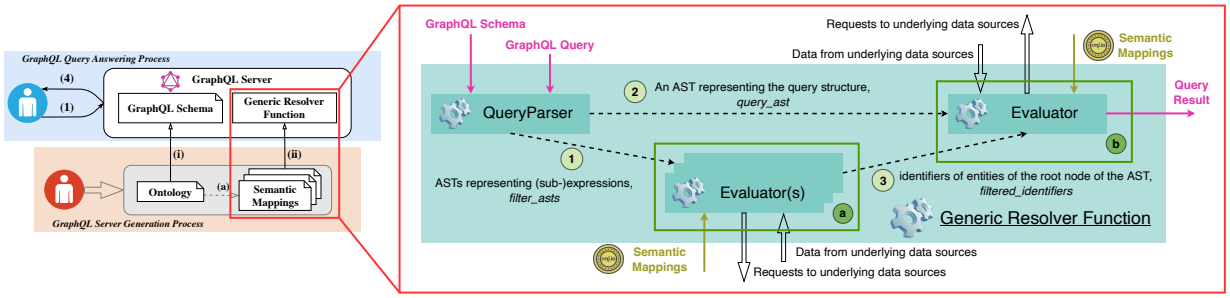


Figure 8. Technical components in the generic resolver function.

Algorithm 2: *Generic Resolver*

Input : a GraphQL query: *query*; a GraphQL schema: *schema*; the semantic mappings: *triples_maps*
Ouput: a list of objects of the type to be queried

```
1  Initialize an empty list: query_result
2  call QueryParser taking query and schema as inputs, to get ASTs: filter_asts, query_ast
3  if filter_asts is not Empty then
       /* there is an input argument given to the query */
4      Initialize an empty set: filtered_identifiers
5      for filter_ast in filter_asts do
6          call Evaluator taking filter_ast and triples_maps as inputs: identifier_info
7          merge filtered_identifiers and identifier_info: filtered_identifiers
8      if filtered_identifiers is not Empty then
9          call Evaluator taking query_ast, triples_maps and filtered_identifiers as inputs: query_result
10     else
11         Do nothing, there is not any instance from data sources satisfying the filter condition.
12 else
       /* there is not an input argument given to the query */
13     call Evaluator taking query_ast, triples_maps as inputs: query_result
14 return query_result
```

retrieves data from the underlying sources of a requested type and relevant fields, it can therefore understand the semantic mappings regarding how to access underlying data sources and structure the returned data according to the GraphQL schema. Taking the query in Figure 3a represented by the ASTs shown in Figure 7 as an example, as the requested type is `University`, the generic resolver function can therefore make use of relevant triples maps (line 1 to line 26 in Listing 3) defined in semantic mappings which are used for transforming underlying data following the semantics related to the *University* concept in the ontology.

5.4. The Evaluator Algorithm

We present the details of *Evaluator* in Algorithm 3 and show an example in Figure 9 of how evaluators work for answering the query in Figure 3a. An AST and a number of triples maps from the semantic mappings are essential inputs to the algorithm. For a given AST, we can obtain the object type and fields that are requested in the query based on the root node and child nodes, respectively (line 2). For instance, taking the ASTs in Figure 7 as examples, the root type and the field for evaluating the filter expression are `University` and `UniversityID`, and the root type and the first level requested field for evaluating query fields are `University` and `departments`, respectively. After getting the relevant triples maps based on the root node type (line 4 in Algorithm 3, e.g., `UniversityMapping` in Listing 3) or from the argument (line 28, the parent triples map, `DepartmentMapping`, which is an argument in the recursive call of an evaluator), the algorithm iterates over triples maps and merges the data obtained based on each triples map (line 5 to line 30). Exploring this in more detail, the algorithm parses each triples map to get the logical source and relevant predicate-object maps (line 8 and line 9). As described in Section 5.2, there are three different types of predicate-object map depending on the different maps of object, which are a reference-valued term map, a constant-valued term map or a referencing-object map. The algorithm iterates over the predicate-object maps and parses each one (line 10 to line 16). For a reference-valued term map, the mapping between the predicate and the reference column or attribute is stored (line 12, e.g., `{UniversityID: uid}` is stored in *pred_attr*), which will be used for rewriting a filter expression according to the underlying data source (line 18, e.g., `uid = 'u1'`), annotating the obtained underlying data (line 21, e.g., `HEAD` is annotated as `head` for *Department* data). For a constant-valued term map, the mapping between the predicate and the constant data value and type is stored (line 14). Both *pred_attr* and *pred_const* will be used to annotate the data from underlying sources (line 21).

In the phase of evaluating a filter expression, *local_filter*, which represents the rewritten filter expression, is a necessary argument when sending requests to underlying data sources (line 19). While in the phase of evaluating query fields, *filter_ids*, being a NULL value or having at least one element, is a necessary argument (line 19, arrow **(a)**

Algorithm 3: *Evaluator*

Input : an Abstract Syntax Tree: *ast*; the semantic mappings: *triples_maps*; the referencing data: *ref*; the identifiers for filtered out result: *filtered_ids*

Output: result of evaluating a filter expression or query fields

```
1  Initialize an empty list: result
2  get the root type and query fields from ast: root_type, query_fields
3  if triples_maps is Empty then
4      get relevant triples maps based on the root_type: triples_maps
5  for tm in triples_maps do
6      Initialize an empty list: referencing_poms
7      Initialize two empty lists: pred_attr, pred_const
8      get the logical source from tm: source
9      get all the predicate-object maps from tm based on query_fields: poms
10     for pom in poms do
11         if object_map in pom is a reference-valued term map then
12             extend pred_attr with a map between the predicate and column/attribute
13         if object_map in pom is a constant-valued term map then
14             extend pred_const with a map between the predicate and data value, type
15         if object_map is a referencing-object map term map then
16             extend referencing_poms with pom
17     parse ast and get the filter expression: filter_expr
18     localize filter_expr based on pred_attr: local_filter
19     access the data source based on source, local_filter, ref, filtered_ids: temp_result
20     if temp_result is not Empty then
21         annotate temp_result based on pred_attr, pred_const
22         for (pred, object_map) in referencing_poms do
23             get the sub tree from ast based on pred: sub_ast
24             parse object_map: parent_triples_map, join_condition
25             parse join_condition: child_field, parent_field
26             get the referencing data from temp_result on child_field: child_data
27             ref = (child_data, parent_field)
28             call Evaluator based on sub_ast, parent_triples_map, ref: parent_data
29             join temp_result and parent_data based on join_condition, pred: temp_result
30     merge result and temp_result: result
31 return result
```

in Figure 9). A NULL value represents the fact that the GraphQL query does not include an input argument. After obtaining the data from the underlying data sources, the data is serialized into JSON format (key/value pairs) in which the keys are predicates stated in the predicate-object map (line 21), where each predicate corresponds to a field in the GraphQL schema. In the next step, the algorithm iterates over predicate-object maps in which the object map refers to another triples map (called a parent triples map) (line 22 to line 29). An evaluator is called again to fetch data based on this parent triples map (line 28, arrow **(4)** in Figure 9). For the query example, the parent triples map refers to the `DepartmentMapping`. Since such a referencing-object map definition states the join condition between the current triples map (`UniversityMapping`) based on *child_field* (`uid`) and the parent triples map (`DepartmentMapping`) based on *parent_field* (`university_id`) (line 21 to line 23 of the mappings in Listing 3), we can pass referencing data (*ref*), which contains the data obtained according to the current triples map and *parent_field*, to the call of an evaluator when we fetch data according to the parent triples map (line 28). Such referencing data is taken into account, in the recursive call to an evaluator, when the request is sent to the underlying data sources (line 19, arrow **(b)** in Figure 9). After the data is obtained according to the parent triples map (arrow **(c)** in Figure 9), it is joined with data obtained according to the current triples map (line 29, frame **(A)** in Figure 9).

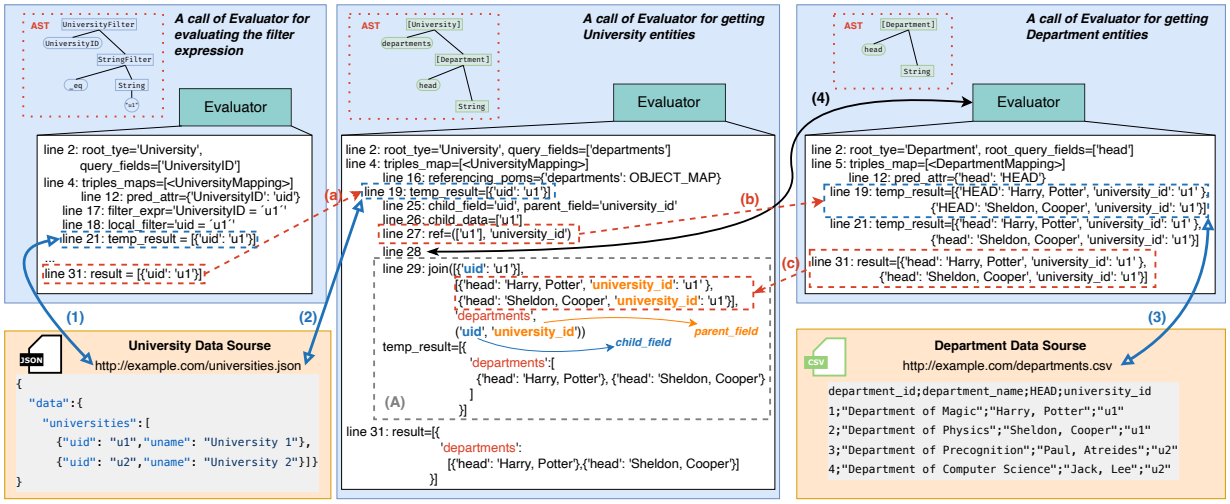


Figure 9. Example for answering the query in Figure 3a, **(1)**-**(3)** indicate the requests to and responses from the data sources; **(a)**-**(c)** indicate the parameter passing between the calls to *Evaluators*; **(4)** indicates a recursive call to *Evaluator* for getting the data of *Departments*; frame **(A)** indicates a join operation.

6. Related Work

The widely used Semantic Web-based techniques and the recently developed GraphQL have led to a number of works relevant to our GraphQL-based framework for data access and data integration. We extend the summary of approaches presented in [30] by adding several new related approaches and new perspectives on the comparison. Table 2 summarizes these systems and our approach. These systems can be divided into two categories, namely OBDA-based systems and GraphQL-based systems. The former group contains Morph-RDB [31, 32], Morph-CSV [33] and Ontop [34, 35]. The latter group consists of GraphQL-LD [36], HyperGraphQL [37], UltraGraphQL [38, 39], Morph-GraphQL [30], Ontology2GraphQL [40] and our OBG-gen.

As a new perspective to the summary in [30], all the approaches (except for GraphQL-LD) have two processes: (i) the service setup (preparation) process and (ii) the query answering process. During the service setup process, some approaches need semantic mappings as input such as Morph-RDB, Morph-CSV, Ontop, Morph-GraphQL and OBG-gen. In such systems, semantic mappings are used in a similar manner to represent differences between global and local schemas. Morph-CSV needs additional annotations for tabular data. OBG-gen needs an ontology and semantic mappings together in order to generate a GraphQL server that is intended not only for semantics-aware data access but for data integration. Morph-GraphQL requires semantic mappings to generate a GraphQL server intended for data access. It does not consider data integration scenarios where integrated views are required. Ontology2GraphQL needs a meta model for the GraphQL query language and requires an ontology following the meta model for generating the GraphQL schema. HyperGraphQL requires no inputs during the service setup process, but the developer must build the GraphQL server from scratch. UltraGraphQL, based on HyperGraphQL, requires RDF schemas of SPARQL endpoints for bootstrapping the GraphQL server. In actuality, GraphQL-LD does not require any GraphQL servers, but instead focuses on how to represent GraphQL queries using SPARQL algebra and how to convert the results of a SPARQL query into a tree structure in response to a GraphQL query.

For the query answering process, OBDA-based approaches (i.e., Morph-RDB, Morph-CSV and Ontop) accept SPARQL queries and translate them into SQL queries. Ontop and Morph-RDB handle underlying data stored in relational databases, while Morph-CSV deals with data stored in CSV files. Our approach, OBG-gen, accepts relational data, CSV-formatted data and JSON-formatted data as the underlying data. The remaining approaches are based on underlying data in SPARQL endpoints and translate GraphQL queries into SPARQL queries. GraphQL-LD, HyperGraphQL, and UltraGraphQL require context information expressed in JSON-LD. Such JSON-LD context information contains URIs of classes to which instances in the RDF data belong.

Table 2
Summary of related approaches.

Approach	Service Setup (Preparation) Process		Query Answering Process		
	Input	Output	Input	Output	Underlying Data
Morph-RDB [31, 32]	semantic mappings	–	SPARQL query	SQL query	Relational data
Morph-CSV [33]	semantic mappings, tabular metadata	–	SPARQL query	SQL query	Tabular data
Ontop [34, 35]	semantic mappings	–	SPARQL query	SQL query	Relational data
GraphQL-LD [36]	–	–	GraphQL query, JSON-LD context	SPARQL query	SPARQL endpoint
HyperGraphQL [37]	–	GraphQL server (manually)	GraphQL query, JSON-LD context	SPARQL query	SPARQL endpoint
UltraGraphQL [38, 39]	RDF schemas of SPARQL endpoints	GraphQL server (automatically)	GraphQL query, JSON-LD context	SPARQL query	SPARQL endpoint
Morph-GraphQL [30]	semantic mappings	GraphQL server (automatically)	GraphQL query	SQL Query	Relational data
Ontology2GraphQL [40]	a meta model, an ontology follows the model	GraphQL server (automatically)	GraphQL query	SPARQL query	SPARQL endpoint
OBG-gen	semantic mappings, an ontology	GraphQL server (automatically)	GraphQL query	SQL query, API requests	Relational data, CSV-formatted data, JSON-formatted data

7. Evaluation

In this section, we present an evaluation of the framework shown in Section 3. We consider a real case application scenario in the materials design domain, and two synthetic benchmark scenarios based on the Linköping GraphQL Benchmark (LinGBM)[6] [41] and GTFS-Madrid-Bench[7] [42], respectively.

The evaluation aims to answer the following research questions:

- **RQ1**: Can the generated GraphQL server provide integrated access to heterogeneous data sources? For instance in the real case application scenario, data from different sources may follow different models and is shared or queries in different ways.
- **RQ2**: How does the generated GraphQL server compare to other OBDA systems and other GraphQL-based systems in terms of query performance and its behavior for increasing dataset sizes?
- **RQ3**: Is the proposed approach, ontology-based GraphQL server generation, a general approach that can work in different domains for data access and integration?

We performed all experiments on a server machine with Intel Xeon Gold 6130 @ 2.10GHz CPUs. The machine runs a 64-bit CentOS Linux 7 (Core) operating system. We reserved 8 CPU cores and 4GB memory for the experiments.

7.1. Real Case Evaluation

In the real case evaluation, we focus on a use case in the materials design domain where the task is data integration over two data sources, Materials Project [43] and OQMD (The Open Quantum Materials Database) [44].

[6] https://github.com/LiUGraphQL/LinGBM
[7] https://github.com/oeg-upm/gtfs-bench

Motivation. The materials science domain, like many other domains, is at an early stage when it comes to introducing Semantic Web-based technologies into its data-driven workflows. A large number of research groups and communities have thus developed a variety of data-driven workflows, including data repositories [45, 46] and data analytics tools. As data-driven techniques become more prevalent, more data is produced by computer programs and is available from various sources, which leads to challenges associated with reproducing, sharing, exchanging, and integrating data among these sources [47–51]. Figure 10 illustrates an example of searching for gallium nitride materials with the reduced chemical formula of GaN in three databases of the materials design domain, Materials Project [43], OQMD [44] and NOMAD (Novel Materials Discovery) [52]. As shown in the results, each of them contains a column that represents chemical composition, but with different column names or different insights (i.e., *'Formula'* for Materials Project and NOMAD, *'Composition'* for OQMD). The *'Formula'* column for Materials Project actually represents the reduced chemical formula. More detailed information regarding the chemical composition can be found based on the value of the *'Nsites'* column. For instance, for the second row of the result from Materials Project, we can derive that the unit cell formula is Ga_2N_2 based on the values of the *'Formula'* and *'Nsites'* columns. Meanwhile, the *'Formula'* column for NOMAD represents the unit cell formula rather than the reduced chemical formula. Unlike the other two databases, OQMD contains a column for reduced chemical formulas, but with a different column name (*'Composition'*). Such differences have to be addressed in order to integrate or exchange data from these data sources. Apart from such differences in terminology, the data that needs to be accessed or integrated from multiple data sources is typically heterogeneous in different models (i.e., relational data stored in relational databases, and hierarchical data stored in JSON data stores). In our previous work, we developed the Materials Design Ontology (MDO) [53] to enable ontology-driven data access and integration. Furthermore, we have applied our MDO and OBG-gen in the OPTIMADE (*Open Database Integrations for Materials Design*) [54] consortium which is a domain effort to make materials databases interoperable. As the work in the consortium is under development, our application is at the level of a proof of concept. Details are presented in [55, p. 141-148].

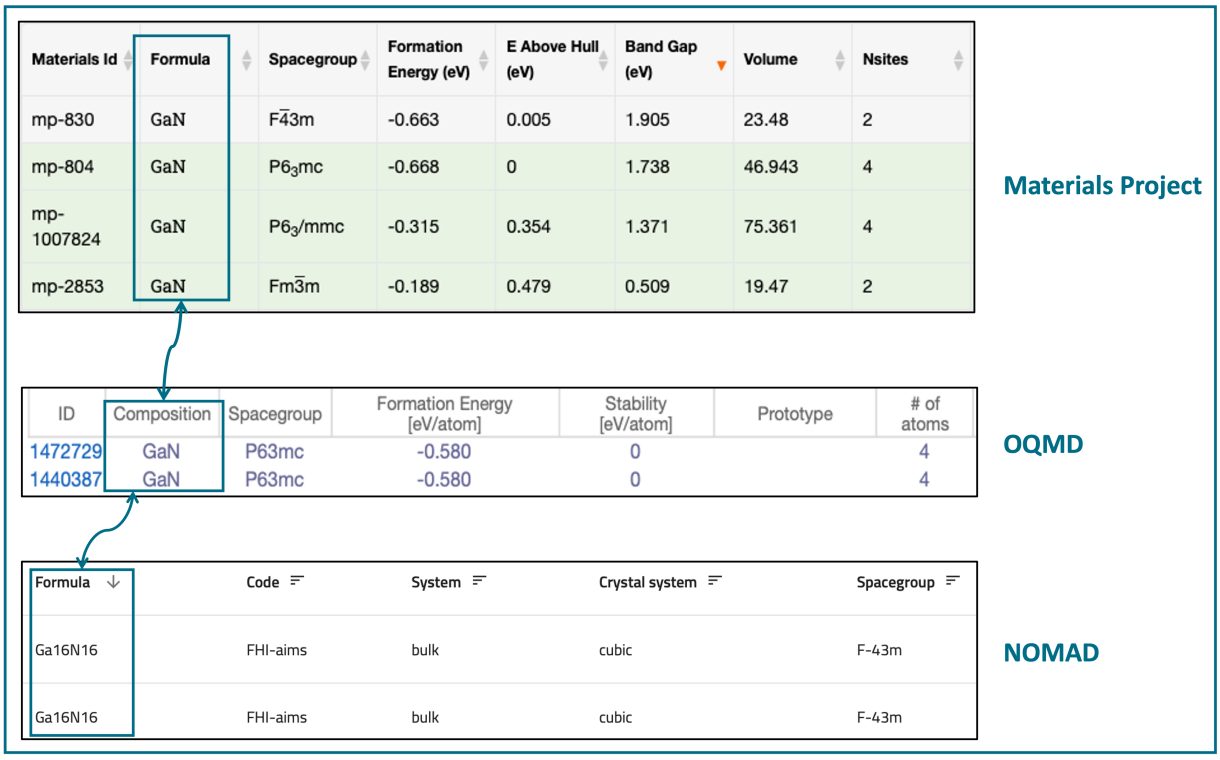


Figure 10. Example of searching materials from Materials Project, OQMD and NOMAD.

Data. We collect data from the Materials Project and OQMD representing five different types of real-world entities (Calculation, Structure, Composition, Band Gap and Formation Energy). We define semantic mappings (for all the systems, see the next paragraph) based on MDO to interpret such data. We collect data in the sizes of 1K, 2K, 4K, 8K, 16K and 32K from each database for populating the five entities. The size 1K means 1000 entities of each entity type. We represent this data in different formats such as tabular data for relational databases and for CSV files, and JSON-formatted data for JSON files. Additionally, for HyperGraphQL and UltraGraphQL in our evaluation, we create an RDF file based on RML mappings and MDO for each dataset setting. We have six dataset settings for the experiments, which are 1K-1K, 2K-2K, 4K-4K, 8K-8K, 16K-16K and 32K-32K. Taking 2K-2K as an example, for each entity type, the test data contains data in the size of 2K from Materials Project and 2K from OQMD, respectively.

Systems. We compare our tool, OBG-gen in two versions (OBG-gen-rdb and OBG-gen-mix) with four systems: Morph-RDB [32], Ontop [35], HyperGraphQL [37], and UltraGraphQL [39]. OBG-gen-rdb represents the case where the generated GraphQL server handles data in relational databases, and OBG-gen-mix represents the case where the generated GraphQL server handles data not only in relational databases but also data in JSON and CSV formats. They take different RML mappings as inputs. Morph-RDB and Ontop are representatives from the group of OBDA-based tools. They access relational databases as data sources by translating SPARQL queries into SQL queries based on semantic mappings. As for the group of GraphQL-related tools, we intended to include Morph-GraphQL and Ontology2GraphQL in our evaluation. However, Morph-GraphQL fails to parse mappings; Ontology2GraphQL cannot be run due to a lack of detailed instructions regarding its setup. In the case of GraphQL-LD, since it focuses on querying Linked Data via GraphQL queries and a JSON-LD context using a SPARQL engine instead of a GraphQL interface, we did not consider it in our evaluation. Therefore, HyperGraphQL and its extension UltraGraphQL are the GraphQL engines that are included in our evaluation. They can query Linked Data that may be provided by local RDF files and remote SPARQL endpoints. The semantic mappings for all the systems in the evaluation are based on MDO. OBG-gen generates the GraphQL schema based on MDO. UltraGraphQL and HyperGraphQL use a modified version of the generated schema since they require directive definitions to specify the correspondences between query entries and the data. Figure 11 shows how the systems are configured in the evaluation. HyperGraphQL and UltraGraphQL are provided with the same RDF data for each dataset setting. OBG-gen-rdb, Morph-RDB and Ontop are provided with two MySQL database instances hosting data from the Materials Project and OQMD respectively. Conceptually, OBG-gen-mix is also provided with two database instances. However, each instance contains different formats of data such as data in a MySQL database, or in CSV or JSON files.

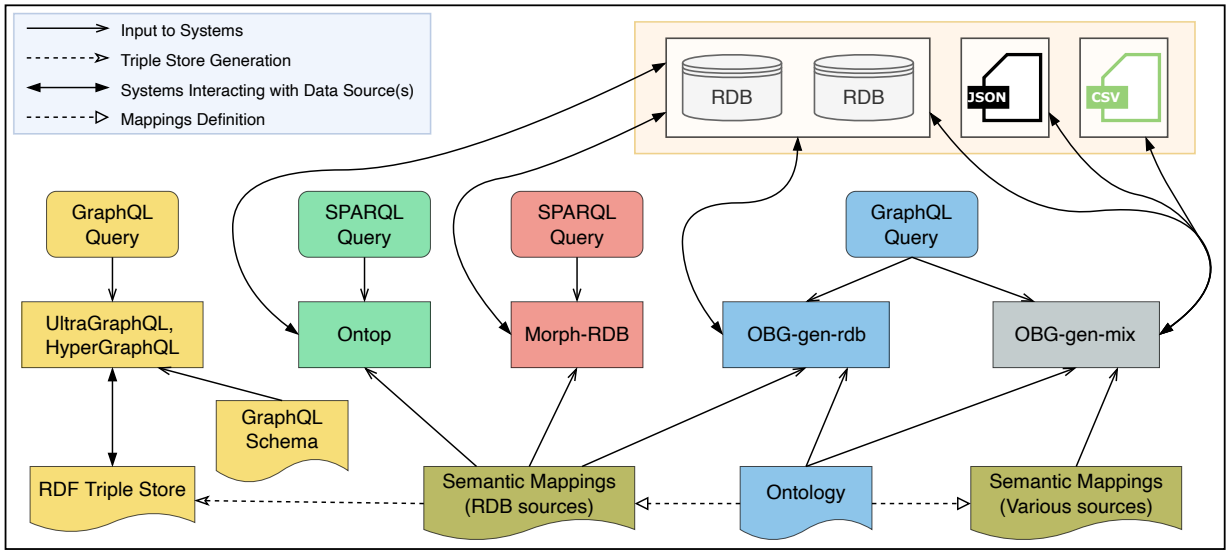


Figure 11. Outline of the real case evaluation.

More detailed, the instance for Materials Project has Composition data in JSON format and Band Gap data in CSV format. The instance for OQMD has Structure and Band Gap data in JSON format and Formation Energy data in CSV format. The data representing other entities for each instance is stored in MySQL database instances.

Queries. We create queries that cover different features, aiming to evaluate our system based on qualitative aspects regarding what functionalities the system can satisfy and quantitative aspects regarding how the system performs over different data sizes. Additionally, we use competency questions stated in the requirements analysis of MDO to create queries with domain interests. Query features of queries without and with filter expressions are shown in Table 3 and Table 4, respectively. From the perspective of GraphQL, we consider which choke point a query covers. The details of choke points are introduced in LinGBM.[8] These choke points are regarding the key technical challenges. We characterize all queries using the perspectives of choke points, domain interest (*DI*), and result size (*RS*). *DI* indicates that the query is a domain-interest query. Such a query corresponds to a relevant competency question stated in the requirements analysis of MDO. For *RS*, as the dataset grows, we consider whether the result size increases linearly (*L*) or more than linearly (*NL*), or stays a constant value (*C*). For queries with filter expressions we take into account the *filter expression form* and whether the filtering AST differs from the query AST (*Diffs*), such as in the example in Figure 7b where the filtering AST and the query AST are different.

Table 5 shows more details of meanings of different filter expressions for Q6–Q12. The filter expressions for Q6 and Q12 are more simple than those for Q7–Q11 where the filter expressions have sub-expressions connected by boolean operators. Query features in terms of *DI*, and the *filter expression form* can help us understand systems qualitatively; *Diffs* and *RS* help in understanding systems quantitatively in the scaling analysis over different data sizes. We show Q1 in Figure 12a and Q7 in Figure 13a. The results of these two queries are given in Figure 12b and Figure 13b, respectively. Q1 requests all the structures containing the reduced chemical formula of each structure composition. Q7 requests all the calculations where the ID is in a given list of values, and the reduced chemical formula is in a given list of values.

Experiments and measurements. We evaluate the query execution time (QET) of the different systems over the six dataset settings. Separately for each query, we run the query four times and always consider the first run to be a warm-up, then take the averaged value of the remaining three runs. Figure 14 illustrates the measurements over the

Table 3
Features of queries without filter conditions.

Query	Choke Points	*Domain Interest (DI)*	*Result Size (RS)*
Q1	2.1, 2.2		*L*
Q2	2.1, 2.2	✓	*L*
Q3	1.1, 2.1, 2.2	✓	*L*
Q4	1.1, 2.1, 2.2	✓	*L*
Q5	2.2		*L*

Table 4
Features of queries with filter conditions.

Query	Choke Points	*Domain Interest (DI)*	*Diffs*	*filter expression form*	*Result Size (RS)*
Q6	1.1, 2.1, 2.2, 4.1, 4.4		✓	A	*C*
Q7	1.1, 2.1, 2.2, 4.1, 4.4		✓	A & B	*C*
Q8	1.1, 2.1, 2.2, 4.1, 4.4, 4.5	✓	✓	A & (B | C)	*C*
Q9	1.1, 2.1, 2.2, 4.1, 4.4, 4.5	✓	✓	A & B	*C*
Q10	1.1, 2.1, 2.2, 4.1, 4.4, 4.5	✓	✓	A & (B & C)	*NL*
Q11	2.2, 4.1, 4.4, 4.5		✓	(A & B) & ((A & B) | C)	*NL*
Q12	2.2, 4.1, 4.4	✓		A	*NL*

[8] https://github.com/LiUGraphQL/LinGBM/wiki/Choke-Points

Table 5
Meanings of filter expressions in Q6 to Q12.

Query	Filter expression meaning
Q6: A	id is in a list
Q7: A & B	id is in a list **and** reduced chemical formula is in a list
Q8: A & (B | C)	id is in a list **and** (reduced chemical formula is in list a_1 **or** list a_2)
Q9: A & B	property name is "Band Gap" **and** value is greater than 5
Q10: A & (B & C)	reduced chemical formula is in a list **and** (property name is "Band Gap" **and** value is greater than 5)
Q11: (A & B) & ((A & B) | C)	(property name is "Band Gap" **and** value is greater than 4) **and** ((property name is "Band Gap" **and** value is greater than 4) **or** reduced chemical formula is in a list)
Q12: A	reduced chemical formula contains silicon element

```
{
  StructureList{
    hasComposition{
      ReducedFormula
    }
  }
}
```

(a) List all the structures containing the reduced chemical formula of each structure's composition.

```
{
  "data": {
    "StructureList": [
      {"hasComposition": {"ReducedFormula": "CeCrS2O"}},
      {"hasComposition": {"ReducedFormula": "TlP(HO2)2"}},
      {"hasComposition": {"ReducedFormula": "YClO"}}
    ]
  }
}
```

(b) The JSON response (an excerpt) of the query.

Figure 12. Example GraphQL query (Q1) in the real case evaluation and the corresponding JSON response (excerpt).

six data sizes per query (Q1–Q12). Figure 15 and Figure 16 illustrate the measurements of all systems per data size for queries without filtering conditions and with filtering conditions, respectively. The measures for all data sizes and all queries are available online.[9] For UltraGraphQL, we have measurements only for queries Q1–Q4 because UltraGraphQL does not support queries with filtering conditions. For HyperGraphQL answering queries with filter expressions, we have only the measurement for Q6 because the system can only deal with filtering by resource IRIs.

Results and discussion. By analyzing the obtained measurements, we summarize three observations. The **first** observation is that both GraphQL servers generated by OBG-gen-rdb and OBG-gen-mix can answer all 12 of the queries covering different features (such as choke points). Therefore, the framework presented in Section 3 is feasible for data access and integration; this answers **RQ1**. Particularly, the GraphQL schema generated based on the ontology can provide an (integrated) view of underlying (heterogeneous) data; the generic resolver function based on the semantic mappings is capable of accessing heterogeneous data sources, combining the retrieved data (which may be in different formats), and structuring the data according to the GraphQL schema.

The **second** observation is regarding queries without filtering conditions (Q1–Q5) (cf. Figures 14 and 15). All of the systems have increases of QETs as the size of the dataset increases. However, Morph-RDB is less sensitive to the data size increase compared with other systems. UltraGraphQL and HyperGraphQL outperform other systems for some smaller datasets (e.g., UltraGraphQL's QETs of Q1 and Q2, HyperGraphQL's QETs for Q1 from 1K-1K to 4K-4K). We explain this by the fact that these two systems have additional context information declaring URIs of classes to which instances in the RDF data belong, which is unlike the other systems which have to make use of semantic mappings to output queries to be evaluated against the underlying data sources. OBG-gen-rdb outperforms Morph-RDB for some queries in smaller datasets (e.g., Q1 in 1K-1K, Q5 in 1K-1K and 2K-2K). For some queries,

[9] https://github.com/LiUSemWeb/OBG-gen/tree/main/evaluation

```
{
  CalculationList(
   filter: {
     _and: [
      {
       ID: {
         _in:["6332","8088","21331",
           "mp-561628","mp-614918"]
       }
      }
      {
        hasOutputStructure: {
          hasComposition: {
            ReducedFormula: {
              _in:["MnCl2","YClO"]
            }
          }
        }
      }
     ]
   }
  )
  {
    ID
    hasOutputCalculatedProperty {
      PropertyName
      numericalValue
    }
  }
}
```

(a) List all the calculations where the ID is in a given list of values and the reduced chemical formula is in a given list of values.

```
{
  "data": {
    "CalculationList": [
      {
        "ID": "6332",
        "hasOutputCalculatedProperty": [
          {
            "PropertyName": "Formation Energy",
            "numericalValue": -1.3247
          },
          {
            "PropertyName": "Band Gap",
            "numericalValue": 1.807
          }
        ]
      },
      {
        "ID": "mp-614918",
        "hasOutputCalculatedProperty": [
          {
            "PropertyName": "Formation Energy",
            "numericalValue": -40.6691
          },
          {
            "PropertyName": "Band Gap",
            "numericalValue": 2.2287
          }
        ]
      }
    ]
  }
}
```

(b) The JSON response (an excerpt) of the query.

Figure 13. Example GraphQL query (Q7) in the real case evaluation and the corresponding JSON response (excerpt).

OBG-gen-rdb and Morph-RDB have close QETs (e.g., Q2 in 1K-1K). Ontop outperforms the other two in smaller datasets (e.g., Q1 in 1K-1K to 8K-8K, Q5 in 1K-1K to 4K-4K), but is more sensitive to data size increase compared with Morph-RDB.

The **third** observation is regarding how OBG-gen-rdb, Ontop and Morph-RDB perform for queries with filter conditions (Q6–Q12) (cf. Figures 14 and 16). Ontop outperforms the other two engines for most cases, but is more sensitive to the change of datasets increase (e.g., Q9 from 1K-1K to 8K-8K). According to [34], Ontop has a mapping optimization step which is not included in the query execution period. This could be a reason why Ontop outperforms the other engines. OBG-gen-rdb and Morph-RDB behave similarly for Q6 with stable QETs and Q12 with slight increases, as the data size increases. As Table 4 shows, the result size of Q6 is a constant over all the datasets in different sizes. Additionally, the filter expressions for Q6 and Q12 are simpler compared with those of Q7–Q11. Therefore, the QETs for evaluating filtering expressions for Q6 and Q12 are less than those of Q7–Q11. For other queries (Q7–Q11) Morph-RDB outperforms OBG-gen-rdb, however the differences between the two systems are less than those for queries without filtering conditions (e.g., Q1–Q4). The filtering conditions in GraphQL queries for OBG-gen-rdb and in SPARQL queries for Morph-RDB are written within WHERE clauses in SQL queries, thus will be evaluated against the back-end databases. A similar observation is also found in [30] where the experiment metrics shows that Morph-RDB outperforms other systems (e.g., Morph-GraphQL) as the size of dataset increase due to the SPARQL to SQL optimizations [30].

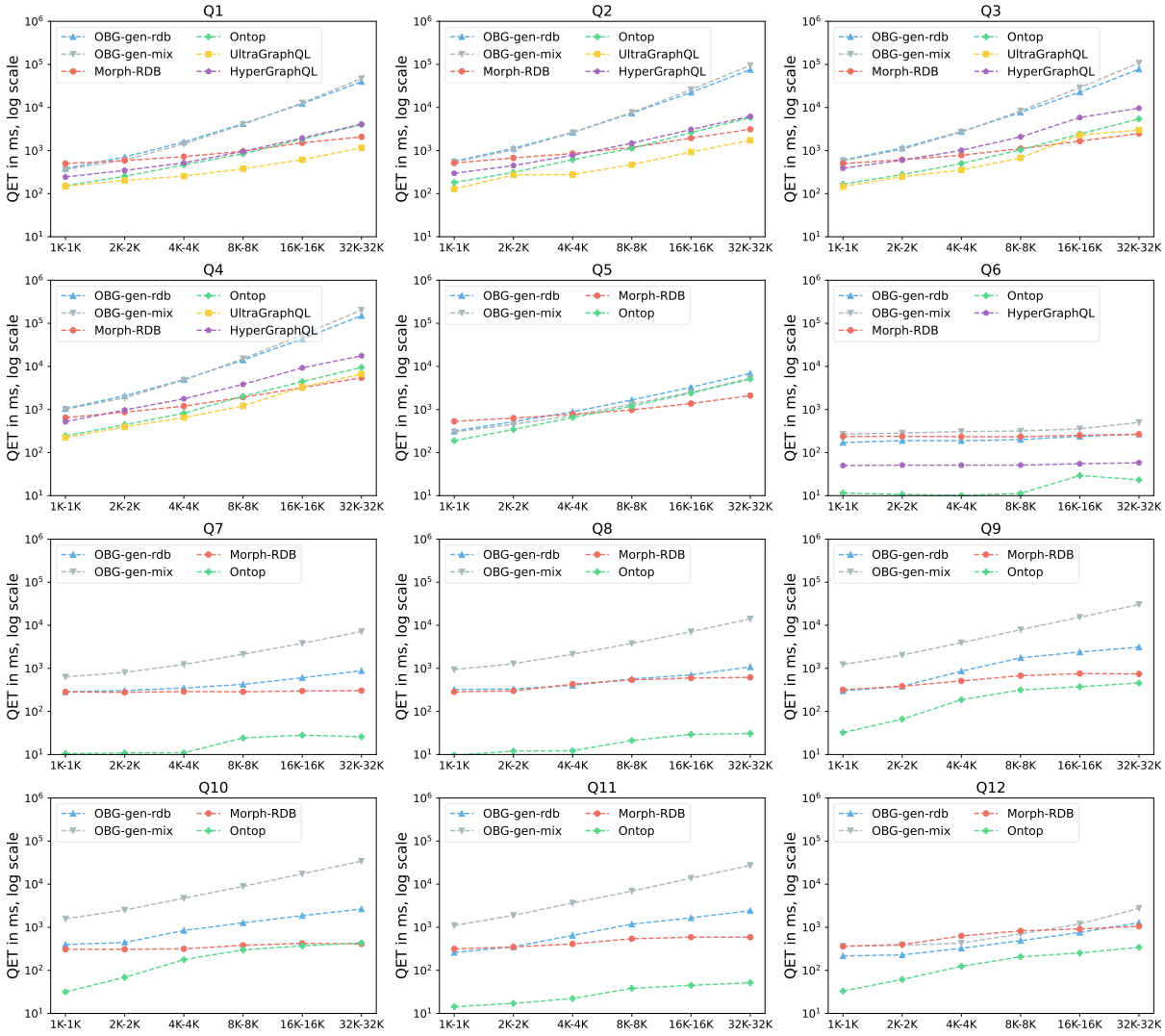


Figure 14. Query Execution Time (QET) per query on materials dataset.

Based on the second and the third observations, we can answer the research question **RQ2**. The GraphQL servers generated by OBG-gen perform similarly compared with other systems for queries without filtering conditions, but are more sensitive to the increase of datasets even they can outperform for some queries in smaller datasets. By comparing OBG-gen-rdb, Ontop and Morph-RDB, we summarize the reasons as follows. As shown in Section 5, the implementation of OBG-gen is based on representing a GraphQL query with Abstract Syntax Trees (cf. Figure 7). In this way, two basic requests are sent to underlying data sources to get the data with respect to the semantic mappings. While for Morph-RDB and Ontop, based on semantic mappings, a SPARQL query is translated to a single SQL query. For queries with filtering conditions, all the three engines (OBG-gen-rdb, Morph-RDB and Ontop) can take the advantages of rewriting filter conditions into SQL queries so that the increases of QETs as data size increases are not obvious.

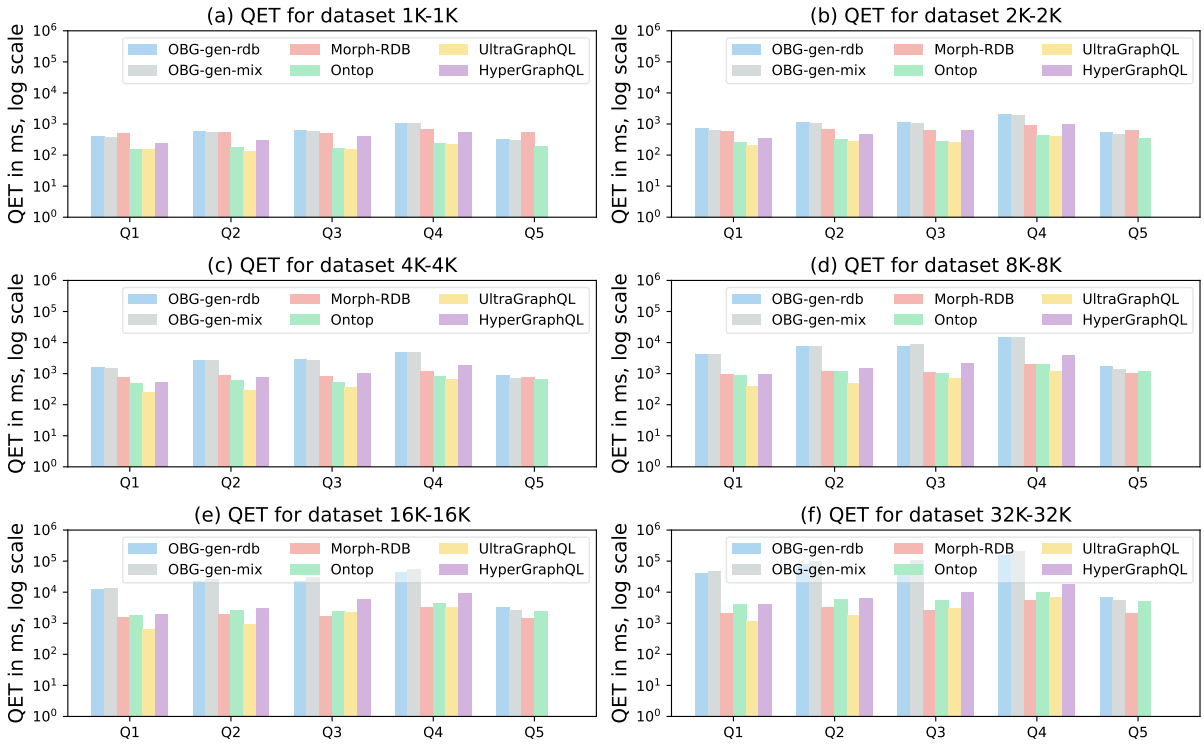


Figure 15. Query Execution Time (QET) per data size on materials dataset for queries without filtering conditions.

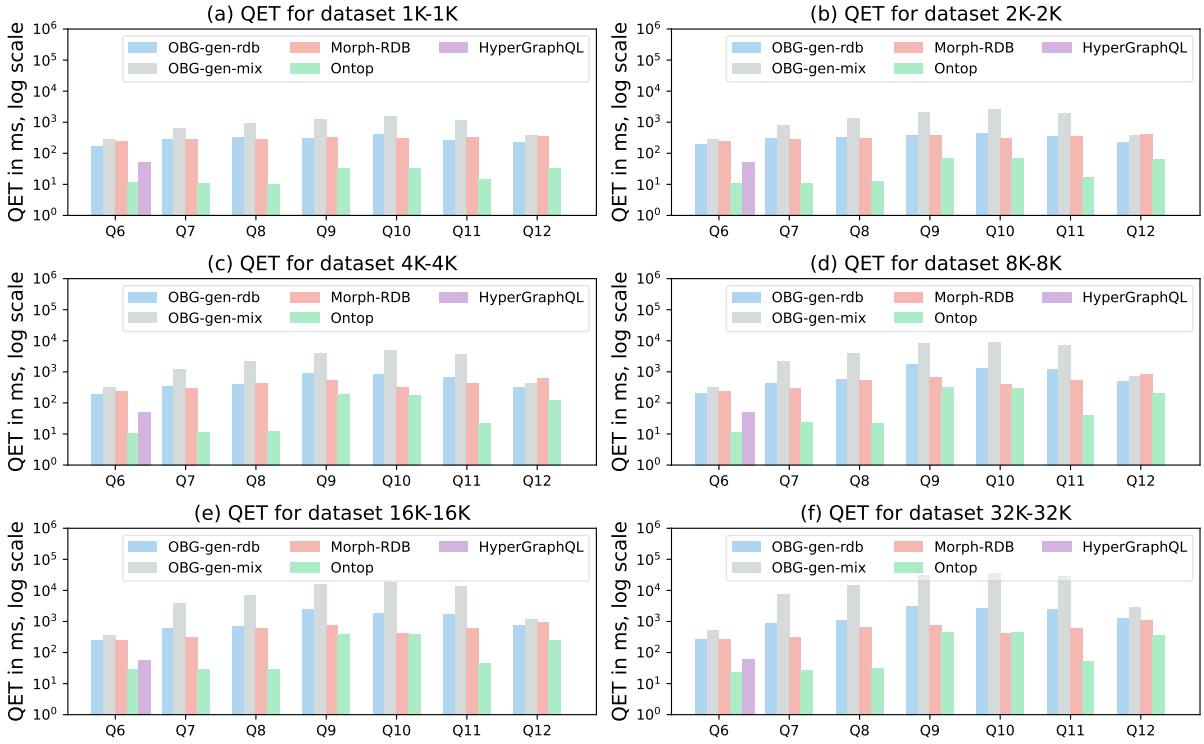


Figure 16. Query Execution Time (QET) per data size on materials dataset for queries with filtering conditions.

7.2. Evaluation based on LinGBM (Linköping GraphQL Benchmark)

To show the generalizability of our system, we conduct an evaluation based on LinGBM. It is developed as a performance benchmark for GraphQL server implementations. LinGBM provides tools for generating datasets (data generator)[10] and queries (query generator),[11] and for testing execution time and response time (test driver).[12]

Data. The dataset generated by the data generator is a scalable, synthetic dataset regarding the *University* domain, including several entity types (e.g., *University* and *Department*). We generate data in scale factors (*sf*) 4, 20 and 100 where a scale factor represents the number of universities [41]. We then create three MySQL database instances to store the data in these three scale factors, respectively. We use a modified version of the GraphQL schema provided by LinGBM for our GraphQL server, and define RML mappings according to the work in Morph-GraphQL.[13] The modification part is regarding input object type definitions so that they can be used to represent filtering conditions.

Queries. The experiments are performed over eight query sets, where each set contains 100 queries that are generated using the LinGBM query generator based on a query template (QT). A query template has placeholders for input arguments. The query generator can generate a set of actual queries (query instances) based on a query template in which the placeholder in the query template is replaced by an actual value. We select eight query templates (QT1–QT6, QT10 and QT11) for constructing eight query sets (QS1–QS8). We show an example query according to QT5 in Listing 4. The other six query templates from LinGBM require GraphQL servers to have implementations for functionalities such as ordering and paging which are not considered currently by OBG-gen. However, these functionalities are interesting for future extension of OBG-gen.

Experiments, results and discussion. Same as the real case evaluation, we evaluate the query execution time (QET) of our system on the three datasets. Each query from a query set is evaluated once. We show the average

Listing 4: A query according to QT5.

```
{
  DepartmentList(filter:{ nr: { _eq: 314 } }) {
    nr
    subOrganizationOf {
      nr
      undergraduateDegreeObtainedBystudent {
        nr
        emailAddress
        memberOf {
          nr
          subOrganizationOf {
            nr
            undergraduateDegreeObtainedBystudent {
              nr
              emailAddress
              memberOf { nr }
            }
          }
        }
      }
    }
  }
}
```

[10] https://github.com/LiUGraphQL/LinGBM/tree/master/tools/datasetgen
[11] https://github.com/LiUGraphQL/LinGBM/tree/master/tools/querygen
[12] https://github.com/LiUGraphQL/LinGBM/tree/master/tools/testdriver_QET_QRT
[13] https://github.com/oeg-upm/morph-graphql/tree/master/examples/LinGBM-v2

Table 6
Average QET (in seconds) in the evaluation based on LinGBM.

Scale Factor	QS1 (QT1)	QS2 (QT2)	QS3 (QT3)	QS4 (QT4)	QS5 (QT5)	QS6 (QT6)	QS7 (QT10)	QS8 (QT11)
4	0.11	0.13	0.12	0.15	0.19	0.13	0.10	0.26
20	0.12	0.15	0.12	0.18	0.51	0.15	0.18	0.90
100	0.15	0.27	0.12	0.26	13.85	0.23	0.72	4.41

query execution times for the different query sets in Table 6. Based on the obtained measurements, we observe that our system has slight increases for QS1, QS2, QS4, QS6 and QS7 in terms of the average QETs. For QS3, the average QET is stable for all the three datasets. For QT5, the increase from 0.51 seconds at data scale factor 20 to 13.85 seconds at data scale factor 100 is due to the dramatic increase in result size. More specifically, the queries in QS5 and QS8 need to access the *'graduateStudent'* table which increases dramatically in size from 50,482 rows in the table (*sf*=20) to 252,562 (*sf*=100). This is the reason for the average QET of QS8 increasing in *sf*=100. Additionally, each query in QS5 repeats a cycle two times (*'university'* to *'graduateStudent'* to *'university'*) and requests the students' emails and addresses along the way. This causes the larger increase in average QET of QS5. The above synthetic experiments indicate that our system can work in another domain than the materials science domain.

7.3. Evaluation based on GTFS-Madrid-Bench

We furthermore demonstrate the generalizability of our system by evaluating it against GTFS-Madrid-Bench, which is a benchmark for evaluating OBDI systems.

Data, queries and systems. The dataset provided by GTFS-Madrid-Bench is a scalable dataset regarding the *Transport* domain (the metro system of Madrid), including several entity types (e.g., *Route*, *Stop*, *Shape* and *Trip*). We use the data generator provided by GTFS-Madrid-Bench to generate data in scale factors (*sf*) 1, 5, 10 and 50. For instance, the dataset in *sf* 1 contains 13 instances for the *Route* type, 1,262 instances for the *Stop* type, 58,540 instances for the *Shape* type and 130 instances for the *Trip* type. An increase in *sf* from 1 to 5 results in an increase in the dataset size of 5 times (e.g., 65 instances for the *Route* type and 6,310 instances for the *Stop* type). Each instance is represented by a row in the corresponding relational table. These scale factors are also used for the experiments in [42]. We then create four MySQL database instances to store the data in these four scale factors, respectively. A total of 18 queries are included in the GTFS-Madrid-Bench benchmark that cover the different features of SPARQL 1.1. For conducting the experiment based on GTFS-Madrid-Bench, we select four queries (Q1–Q4) to create corresponding GraphQL queries. Among these four queries, Q1 retrieves all the shape entities where each shape entity is a polygon associated with a trip; Q2 retrieves all the stop entities where the latitude is greater than a specific value; Q3 retrieves accessibility information of all stop entities; Q4 retrieves all the route entities and their associated agency entities [42]. The other queries contain SPARQL 1.1 features such as order by, group by and distinct. Currently, OBG-gen does not implement functionalities to cover these features. However, these functionalities are interesting for future extension of OBG-gen. In addition to OBG-gen-rdb, we conduct experiments based on Ontop to learn how two engines behave in this GTFS-Madrid-Bench benchmark scenario.

Experiments, results and discussion. Same as the previous two evaluation scenarios, we evaluate the query execution time (QET) of systems on different datasets. We show the measurements in Table 7. According to the measurements, both OBG-gen-rdb and Ontop show increases in QETs for all four queries as the dataset increases. However, as with the observation in the real case evaluation, Ontop behaves less sensitively to the increase in dataset. In terms of how the two systems behave for different queries, both engines spend more time to answer Q1 (without any filter conditions). It takes OBG-gen-rdb more than 3,600 seconds to answer it for scale factors 10 and 50. Although Ontop is able to answer Q1 in less time than OBG-gen, it cannot finish the execution because it runs out of the reserved 4GB memory for scale factor 50. More specifically, Q1 needs to access the *'Shape'* table which increases dramatically in size from 58,540 rows in the table (*sf*=1) to 292,700 (*sf*=5) and furthermore to 585,400 (*sf*=10) and 2,927,000 (*sf*=50). Both engines have relatively stable QETs for Q4 that retrieves all the route entities without any filter conditions. The corresponding *'Route'* table is relatively small (e.g., 13 for *sf* 1 and 1,300 for *sf* 100). Q2 and

Table 7
QET (in seconds) in the evaluation based on GTFS-Madrid-Bench.

Scale Factor	System	Q1	Q2	Q3	Q4
1	OBG-gen-rdb	82.22	0.373	0.337	0.085
	Ontop	7.311	0.132	0.100	0.014
5	OBG-gen-rdb	2610	1.743	1.413	0.115
	Ontop	37.596	0.480	0.384	0.030
10	OBG-gen-rdb	time out	4.700	3.030	0.143
	Ontop	75.072	0.939	0.703	0.048
50	OBG-gen-rdb	time out	84.159	50.125	0.255
	Ontop	out of memory	8.052	4.044	0.155

Q3 retrieve all the station entities but with different filter conditions. The two engines spend more time to evaluate Q2 since the result size of Q2 is larger than that of Q3.

7.4. Summary and Discussion

For evaluating our approach, ontology-based GraphQL server generation, we conducted an experiment motivated by the materials design domain and experiments based on two synthetic benchmark scenarios (LinGBM and GTFS-Madrid-Bench). Based on the measurements of these experiments, we can answer the three research questions presented at the beginning of Section 7. Our approach can generate GraphQL servers for data access and data integration and can be used in various domains (**RQ1** and **RQ3**). The other GraphQL interfaces, HyperGraphQL and UltraGraphQL, can be used for data integration to a limited extent due to the fact that they do not support various filter conditions. This means questions with filter conditions cannot be answered. By comparing our approach with other well-known systems (e.g., Morph-RDB and Ontop), we learn that our system can perform relatively similar to others in terms of QETs for queries with filter conditions, and for some queries without filter conditions in smaller datasets (**RQ2**). Morph-RDB and Ontop are both less sensitive to the data size increase. The reason for this can be explained by the fact that they have optimization techniques that enable queries to be executed in a shorter amount of time. For instance, Ontop has a mapping optimization step which is not included in the query execution period [34]; Morph-RDB has a query rewriting optimization step where projections and selections are pushed down for removing non-correlated subqueries [31]. However, our approach supports data integration where the underlying data is from different kinds of sources (i.e., OBG-gen-mix), in contrast to Morph-RDB and Ontop that only support data integration where the underlying data is from relational databases.

8. Concluding Remarks and Future Work

To leverage ontologies for generating GraphQL APIs to support semantics-aware data access and data integration, in this paper, we have presented a GraphQL-based framework (cf. Section 3) for data access and integration in which an ontology drives the generation of the GraphQL server. Our approach consists of a formal method to generate a GraphQL schema based on an ontology (cf. Section 4), and a generic implementation of resolver functions (cf. Section 5). In detail, ontologies play two roles in our approach: one is as an integrated view of underlying data sources for generating a GraphQL schema; the other is as a basis for defining semantic mappings on which the generic GraphQL resolver function is based. Generating a GraphQL schema based on an ontology rather than just semantic mappings (e.g., Morph-GraphQL) can ensure to have an integrated view of data in data integration scenarios. Such a schema does not need to be regenerated when new data sources are added, unless the ontology needs to be modified. We show the feasibility and usefulness of our approach in terms of using GraphQL for data integration and avoiding implementing a GraphQL server from scratch, based on a real-world data integration scenario motivated by the materials design domain (cf. Section 7.1) and two synthetic benchmark scenarios, LinGBM and GTFS-Madrid-Bench (cf. Sections 7.2 and 7.3).

Our current effort of OBG-gen focuses on the current GraphQL language features that support semantics-aware and integrated data access. In the future we will follow the development of the GraphQL language and investigate

if any new features for data access can be generated formally based on the description logic currently used by OBG-gen or whether a more expressive language is needed. Additionally, optimizing our generic resolver function to improve query performance (e.g., adapting the *mapping partition group rules* recently proposed in [56]) and extending our approach to support various query features (e.g., order by, group by) are interesting directions for future work. Furthermore, from a practical point of view, we will implement a search system for OPTIMADE, based on our approach, in the materials design domain.

Acknowledgements. This work has been financially supported by the Swedish e-Science Research Centre (SeRC), the Swedish National Graduate School in Computer Science (CUGS), the Swedish Research Council (Vetenskapsrådet, dnr 2018-04147 and dnr 2019-05655), and the Swedish Agency for Economic and Regional and Growth (Tillväxtverket).

References

[1] Facebook Inc., Specification for GraphQL-June 2020 Edition. http://spec.graphql.org/draft/.
[2] B. Bhattacharya, An introduction to GraphQL federation, Accessed: 2022-07-29. https://tyk.io/blog/an-introduction-to-graphql-federation/.
[3] Apollo Inc., Introduction to Apollo Federation, Accessed: 2022-07-29. https://www.apollographql.com/docs/federation/.
[4] B. Smith, *Ontology*, in: *The furniture of the world*, Brill, 2012, pp. 47–68. doi:10.1163/9789401207799_005.
[5] C. Welty, Ontology Research, *AI Magazine* **24**(3) (2003), 11. doi:10.1609/aimag.v24i3.1714.
[6] R. Studer, V.R. Benjamins and D. Fensel, Knowledge engineering: Principles and methods, *Data & Knowledge Engineering* **25**(1) (1998), 161–197. doi:10.1016/S0169-023X(97)00056-6.
[7] R. Stevens, C.A. Goble and S. Bechhofer, Ontology-based Knowledge Representation for Bioinformatics, *Briefings in Bioinformatics* **1**(4) (2000), 398–414. doi:10.1093/bib/1.4.398.
[8] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi and P.F. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications*, 2nd edn, Cambridge University Press, 2007. doi:10.1017/CBO9780511711787.
[9] M. Lenzerini, Data Integration: A Theoretical Perspective, in: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, Association for Computing Machinery, 2002, pp. 233–246. doi:10.1145/543613.543644.
[10] D. Calvanese and G. De Giacomo, Data Integration: A Logic-Based Perspective, *AI magazine* **26**(1) (2005), 59–59. doi:10.1609/aimag.v26i1.1799.
[11] A. Doan, A. Halevy and Z. Ives, 1 - Introduction, in: *Principles of Data Integration*, Morgan Kaufmann, 2012, pp. 1–18. doi:10.1016/B978-0-12-416044-6.00001-6.
[12] D. Calvanese, G.D. Giacomo, D. Lembo, M. Lenzerini and R. Rosati, *Ontology-Based Data Access and Integration*, in: *Encyclopedia of Database Systems*, Springer, New York, NY, 2018, pp. 2590–2596. doi:10.1007/978-1-4614-8265-9_80667.
[13] G. Xiao, D. Hovland, D. Bilidas, M. Rezk, M. Giese and D. Calvanese, Efficient Ontology-Based Data Integration with Canonical IRIs, in: *The Semantic Web 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings*, Lecture Notes in Computer Science, Vol. 10843, Springer,Cham, 2018, pp. 697–713. doi:10.1007/978-3-319-93417-4_45.
[14] G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati and M. Zakharyaschev, Ontology-Based Data Access: A Survey, in: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, International Joint Conferences on Artificial Intelligence Organization, 2018, pp. 5511–5519. doi:10.24963/ijcai.2018/777.
[15] O. Corcho, F. Priyatna and D. Chaves-Fraga, Towards a new generation of ontology based data access, *Semantic Web* **11**(1) (2020), 153–160. doi:10.3233/SW-190384.
[16] G. Wiederhold, Mediators in the Architecture of Future Information Systems, *Computer* **25**(3) (1992), 38–49. doi:10.1109/2.121508.
[17] P. Vassiliadis, A Survey of Extract-Transform-Load Technology, *Integrations of Data Warehousing, Data Mining and Database Technologies: Innovative Approaches* **5**(3) (2009), 1–27. doi:10.4018/978-1-60960-537-7.ch008.
[18] S. Das, S. Sundara and R. Cyganiak, R2RML: RDB to RDF Mapping Language, Accessed: 2022-07-29. https://www.w3.org/TR/r2rml/.
[19] M. Arenas, A. Bertails, E. Prud'hommeaux and J. Sequeda, A Direct Mapping of Relational Data to RDF, Accessed: 2022-07-29. https://www.w3.org/TR/rdb-direct-mapping/.
[20] 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: *Proceedings of the Workshop on Linked Data on the Web co-located with the 23rd International World Wide Web Conference (WWW 2014)*, CEUR Workshop Proceedings, Vol. 1184, 2014. http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.
[21] A. Dimou, M. Vander Sande, J. Slepicka, P. Szekely, E. Mannens, C. Knoblock and R. Van de Walle, Mapping Hierarchical Sources into RDF Using the RML Mapping Language, in: *2014 IEEE International Conference on Semantic Computing*, 2014, pp. 151–158. doi:10.1109/ICSC.2014.25.

[22] O. Hartig and J. Hidders, Defining Schemas for Property Graphs by Using the GraphQL Schema Definition Language, in: *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) co-located with SIGMOD/PODS '19: International Conference on Management of Data*, GRADES-NDA'19, Association for Computing Machinery, 2019, pp. 1–11. doi:10.1145/3327964.3328495.

[23] O. Hartig and J. Pérez, Semantics and Complexity of GraphQL, in: *Proceedings of the 2018 World Wide Web Conference*, WWW '18, 2018, pp. 115–1164. doi:10.1145/3178876.3186014.

[24] F. Baader, O.F. Gil and M. Pensel, Standard and Non-Standard Inferences in the Description Logic FL_0 Using Tree Automata., in: *GCAI-2018. 4th Global Conference on Artificial Intelligence*, Vol. 55, 2018, pp. 1–14. doi:10.29007/scbw.

[25] I. Horrocks, P.F. Patel-Schneider and F. Van Harmelen, From SHIQ and RDF to OWL: The making of a web ontology language, *Journal of Web Semantics* **1**(1) (2003), 7–26. doi:10.1016/j.websem.2003.07.001.

[26] I. Horrocks and U. Sattler, Ontology Reasoning in the SHOQ(D) Description Logic, in: *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'01, Morgan Kaufmann Publishers Inc., 2001, pp. 199–204–.

[27] F. Baader, I. Horrocks, C. Lutz and U. Sattler, *An Introduction to Description Logic*, 1st edn, Cambridge University Press, 2017. doi:10.1017/9781139025355.

[28] F. Baader, P. Marantidis and M. Pensel, The Data Complexity of Answering Instance Queries in FL_0, in: *Companion Proceedings of the The Web Conference 2018*, WWW '18, International World Wide Web Conferences Steering Committee, 2018, pp. 1603–1607. doi:10.1145/3184558.3191618.

[29] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini and R. Rosati, Linking Data to Ontologies, in: *Journal on Data Semantics X*, Springer, 2008, pp. 133–173. doi:10.1007/978-3-540-77688-8_5.

[30] D. Chaves-Fraga, F. Priyatna, A. Alobaid and O. Corcho, Exploiting Declarative Mapping Rules for Generating GraphQL Servers with Morph-GraphQL, *International Journal of Software Engineering and Knowledge Engineering* **30**(06) (2020), 785–803. doi:10.1142/S0218194020400070.

[31] F. Priyatna, O. Corcho and J. 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*, Association for Computing Machinery, 2014, pp. 479–490. doi:10.1145/2566486.2567981.

[32] Morph-RDB, version 3.12.5, Accessed: 2022-07-29. https://github.com/oeg-upm/morph-rdb/releases/tag/v3.12.5.

[33] D. Chaves-Fraga, E. Ruckhaus, F. Priyatna, M. Vidal and O. Corcho, Enhancing Virtual Ontology Based Access over Tabular Data with Morph-CSV, *Semantic Web* **12**(6) (2021), 869–902. doi:10.3233/SW-210432.

[34] 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. doi:10.3233/SW-160217.

[35] Ontop, version 4.2.1, Accessed: 2022-07-29. https://github.com/ontop/ontop/releases/tag/ontop-4.2.1.

[36] R. Taelman, M. Vander Sande and R. Verborgh, GraphQL-LD: Linked Data Querying with GraphQL, in: *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018)*, CEUR Workshop Proceedings, Vol. 2180, CEUR-WS, 2018. http://ceur-ws.org/Vol-2180/paper-65.pdf.

[37] Semantic Integration Ltd., HyperGraphQL, version 2.0.0, Accessed: 2022-07-29. https://github.com/hypergraphql/hypergraphql/releases/tag/2.0.0.

[38] L. Gleim, T. Holzheim, I. Koren and S. Decker, Automatic Bootstrapping of GraphQL Endpoints for RDF Triple Stores, in: *Proceedings of the QuWeDa 2020: 4th Workshop on Querying and Benchmarking the Web of Data co-located with 19th International Semantic Web Conference (ISWC 2020)*, CEUR Workshop Proceedings, Vol. 2722, CEUR-WS, 2020, pp. 119–134. http://ceur-ws.org/Vol-2722/quweda2020-paper-2.pdf.

[39] Semantic Integration Ltd., UltraGraphQL, version 1.0.0, Accessed: 2022-07-29. https://git.rwth-aachen.de/i5/ultragraphql.

[40] C. Farré, J. Varga and R. Almar, GraphQL Schema Generation for Data-Intensive Web APIs, in: *Model and Data Engineering*, Springer, Cham, 2019, pp. 184–194. doi:10.1007/978-3-030-32065-2_13.

[41] S. Cheng and O. Hartig, LinGBM: A Performance Benchmark for Approaches to Build GraphQL Servers, in: *Web Information Systems Engineering – WISE 2022 - 23rd International Conference on Web Information Systems Engineering, Biarritz, France, November 1–3, 2022*, Lecture Notes in Computer Science, Vol. 13724, Springer, Cham, 2022. doi:10.1007/978-3-031-20891-1_16.

[42] D. Chaves-Fraga, F. Priyatna, A. Cimmino, J. Toledo, E. Ruckhaus and O. Corcho, GTFS-Madrid-Bench: A benchmark for virtual knowledge graph access in the transport domain, *Journal of Web Semantics* **65** (2020), 100596. doi:10.1016/j.websem.2020.100596.

[43] A. Jain, S.P. Ong, G. Hautier, W. Chen, W.D. Richards, S. Dacek, S. Cholia, D. Gunter, D. Skinner, G. Ceder and K.a. Persson, The Materials Project: A materials genome approach to accelerating materials innovation, *APL Materials* **1**(1) (2013), 011002. doi:10.1063/1.4812323.

[44] J.E. Saal, S. Kirklin, M. Aykol, B. Meredig and C. Wolverton, Materials Design and Discovery with High-Throughput Density Functional Theory: The Open Quantum Materials Database (OQMD), *JOM* **65** (2013), 1501–1509. doi:10.1007/s11837-013-0755-4.

[45] P. Lambrix, R. Armiento, A. Delin and H. Li, Big Semantic Data Processing in the Materials Design Domain, in: *Encyclopedia of Big Data Technologies*, Springer, 2019. doi:10.1007/978-3-319-63962-8_293-1.

[46] P. Lambrix, R. Armiento, A. Delin and H. Li, FAIR Big Data in the Materials Design Domain, in: *Encyclopedia of Big Data Technologies*, Springer, 2022. doi:10.1007/978-3-319-63962-8_293-2.

[47] S.R. Kalidindi and M. De Graef, Materials Data Science: Current Status and Future Outlook, *Annual Review of Materials Research* **45** (2015), 171–193. doi:10.1146/annurev-matsci-070214-020844.

[48] A. Agrawal and A. Choudhary, Perspective: Materials informatics and big data: Realization of the "fourth paradigm" of science in materials science, *APL Materials* **4** (2016), 053208:1–10. doi:10.1063/1.4946894.

[49] A. Tropsha, K.C. Mills and A.J. Hickey, Reproducibility, sharing and progress in nanomaterial databases, *Nature Nanotechnology* **12** (2017), 1111–1114. doi:10.1038/nnano.2017.233.

[50] S. Karcher, E.L. Willighagen, J. Rumble, F. Ehrhart, C.T. Evelo, M. Fritts, S. Gaheen, S.L. Harper, M.D. Hoover, N. Jeliazkova, N. Lewinski, R.L.M. Robinson, K.C. Mills, A.P. Mustad, D.G. Thomas, G. Tsiliki and C.O. Hendren, Integration among databases and data sets to support productive nanotechnology: Challenges and recommendations, *NanoImpact* **9** (2018), 85–101. doi:10.1016/j.impact.2017.11.002.

[51] J. Rumble, J. Broome and S. Hodson, Building an International Consensus on Multi-Disciplinary Metadata Standards: A CODATA Case History in Nanotechnology, *Data Science Journal* **8** (2019), 12:1–11. doi:10.5334/dsj-2019-012.

[52] C. Draxl and M. Scheffler, NOMAD: The FAIR concept for big data-driven materials science, *MRS Bulletin* **43**(9) (2018), 676–682. doi:10.1557/mrs.2018.208.

[53] H. Li, R. Armiento and P. Lambrix, An Ontology for the Materials Design Domain, in: *The Semantic Web - ISWC 2020 - 19th International Semantic Web Conference, Athens, Greece, November 2-6, 2020*, Lecture Notes in Computer Science, Vol. 12507, Springer, Cham, 2020, pp. 212–227. doi:10.1007/978-3-030-62466-8_14.

[54] C.W. Andersen, R. Armiento, E. Blokhin, G.J. Conduit, S. Dwaraknath, M.L. Evans, Á. Fekete, A. Gopakumar, S. Gražulis, A. Merkys, F. Mohamed, C. Oses, G. Pizzi, G.-M. Rignanese, M. Scheidgen, L. Talirz, C. Toher, D. Winston, R. Aversa, K. Choudhary, P. Colinet, S. Curtarolo, D. Di Stefano, C. Draxl, S. Er, M. Esters, M. Fornari, M. Giantomassi, M. Govoni, G. Hautier, V. Hegde, M.K. Horton, P. Huck, G. Huhs, J. Hummelshøj, A. Kariryaa, B. Kozinsky, S. Kumbhar, M. Liu, N. Marzari, A.J. Morris, A.A. Mostofi, K.A. Persson, G. Petretto, T. Purcell, F. Ricci, F. Rose, M. Scheffler, D. Speckhard, M. Uhrin, A. Vaitkus, P. Villars, D. Waroquiers, C. Wolverton, M. Wu and X. Yang, OPTIMADE: an API for exchanging materials data, *Scientific Data* **8**(217) (2021). doi:10.1038/s41597-021-00974-z.

[55] H. Li, Ontology-Driven Data Access and Data Integration with an Application in the Materials Design Domain, PhD thesis, Linköping University, Sweden, 2022. doi:10.3384/9789179292683.

[56] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M.S. Pérez and O. Corcho, Morph-KGC: Scalable Knowledge Graph Materialization with Mapping Partitions, *Semantic Web* (2022). doi:10.3233/SW-223135.