OBG-gen: Ontology-Based GraphQL Server Generation for Data Integration

Huanyu Li[1,2,*], Olaf Hartig[1], Rickard Armiento[2,3] and Patrick Lambrix[1,2]

[1]Department of Computer and Information Science, Linköping University, Linköping, Sweden
[2]Swedish e-Science Research Centre, Linköping, Sweden
[3]Department of Physics, Chemistry and Biology, Linköping University, Linköping, Sweden

Abstract

A GraphQL server contains two building blocks: (1) a GraphQL schema defining the types of data objects that can be requested; (2) resolver functions fetching the relevant data from underlying data sources. GraphQL can be used for data integration if the GraphQL schema provides an integrated view of data from multiple data sources, and the resolver functions are implemented accordingly. However, there does not exist a semantics-aware approach to use GraphQL for data integration. We proposed a framework using GraphQL for data integration in which a global domain ontology informs the generation of a GraphQL server. Furthermore, we implemented a prototype of this framework, OBG-gen. In this paper, we demonstrate OBG-gen in a real-world data integration scenario in the materials design domain and in a synthetic benchmark scenario.

Keywords

GraphQL, Ontology, Data Integration, GraphQL Server Generation

1. Introduction

GraphQL[1] is a conceptual framework for building Web APIs. The framework introduces a so-called GraphQL schema (Figure 1a) that defines the types of data objects that can be requested, and resolver functions (Figure 1b) that specify how to retrieve and fetch data from underlying data sources. Another building block of the framework is the GraphQL query language for expressing data retrieval requests (Figure 1c). The example schema contains an object type (`University`) with a field definition `UniversityID` of which the value type is `String` and a field definition `departments` of which the value type is `[Department]`. It also contains two input object types (`UniversityFilter` and `StringFilter`) which can capture the notions of filter expressions. For instance, the query accepts the argument (`UniversityID:{_eq: "u1"}`) according to the

ISWC 2023 Posters and Demos: 22nd International Semantic Web Conference, November 6–10, 2023, Athens, Greece
*Corresponding author.
huanyu.li@liu.se (H. Li); olaf.hartig@liu.se (O. Hartig); rickard.armiento@liu.se (R. Armiento); patrick.lambrix@liu.se (P. Lambrix)
http://huanyuli.se (H. Li); https://olafhartig.de (O. Hartig); https://rickard.armiento.se (R. Armiento); https://www.ida.liu.se/~patla00/ (P. Lambrix)
0000-0003-1881-3969 (H. Li); 0000-0002-1741-2090 (O. Hartig); 0000-0002-5571-0814 (R. Armiento); 0000-0002-9084-0470 (P. Lambrix)

[1]http://spec.graphql.org/October2021/

```
type University {
  UniversityID: String
  departments: [Department]  }
input UniversityFilter {
  UniversityID: StringFilter
  _and: [UniversityFilter]  }
input StringFilter {
  _eq: String
  _in: [String]  }
type Query {
  UniversityList(filter:
   UniversityFilter): [University] }
```

(a) GraphQL schema example.

```
const UniversityList = (uid) => {
  /*assume the underlying data source is a
    relational database containing a table
    named university with an id column*/
  let data = db_connection.select().from(
       'university').where('id', uid);
  /*assume University is an object defined
    according to the type in the schema*/
  let allUniversities = data.then(rows =>
       new University(rows[0]));
  return allUniversities;
};
```

(b) Resolver function example.

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

(c) Query example.

Figure 1: Example GraphQL schema, resolver function and query.

`UniversityFilter` definition, which represents "*UniversityID is equal to 'u1'*". Additionally, GraphQL schema presumes the `Query` type as the query root operation type. The example schema has the `UniversityList` field definition of which the returned type is `[University]`, a list of universities. GraphQL can be used for data integration by building a GraphQL server over underlying data sources where the GraphQL schema provides an integrated view of data, and the resolver functions specify implementations for accessing data sources. However, there does not exist a semantics-aware approach to employ GraphQL for data integration. It means the developer needs to write program code (i.e., resolver functions) to populate the various elements of a GraphQL schema. In our previous work, we provided a semantics-aware approach to employ GraphQL for data integration which is a global as view approach [1], with formal methods to generate the GraphQL server [2]. In this paper, we demonstrate the implemented prototype of this approach, OBG-gen.[2]

2. Approach

Figure 2 shows the framework of our approach for data integration based on GraphQL. The framework relies on an ontology as an integrated view of the data from multiple sources, and semantic mappings defining how the underlying data can be interpreted by the global ontology (arrows (**a**) and (**b**)). In addition, there are two processes defined in the framework. The first process is for automatically generating the GraphQL schema (arrow (**i**)) and implementing a generic resolver function based on semantic mappings (arrow (**ii**)). This process thus can benefit GraphQL application developers by eliminating constructing GraphQL servers from scratch. The second process is for answering (integrated) GraphQL queries (arrow (**1**) to (**4**)).

To generate the GraphQL schema based on an ontology, we assume that the ontology is represented by a TBox in a description logic which allows atomic concepts, the universal concept, intersection, value restriction, qualified number restrictions and datatypes. The general concept inclusion (GCI) of the TBox can be five forms: $P \sqsubseteq Q$, $P \sqsubseteq \forall r.Q$, $P \sqsubseteq = 1r.Q$, $P \sqsubseteq \forall a.d$, $P \sqsubseteq = 1a.d$ where P and Q are atomic concepts, r is a role, a is an attribute and d is a datatype.

[2]All the material related to OBG-gen is available online at https://github.com/LiUSemWeb/OBG-gen.

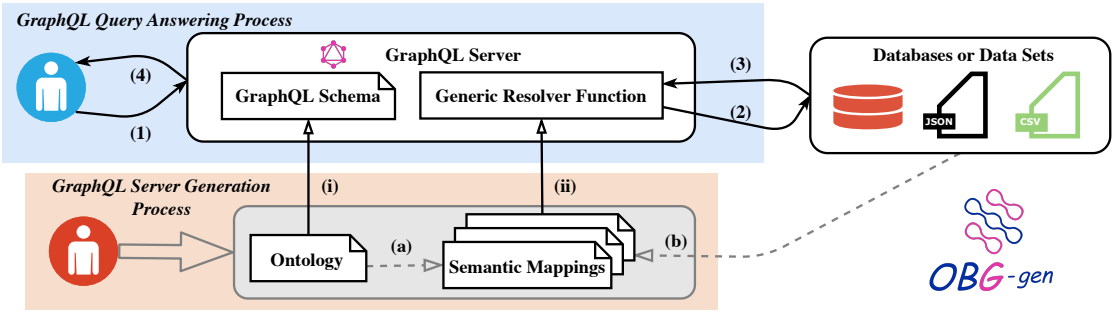


Figure 2: GraphQL-based framework for data access and integration.

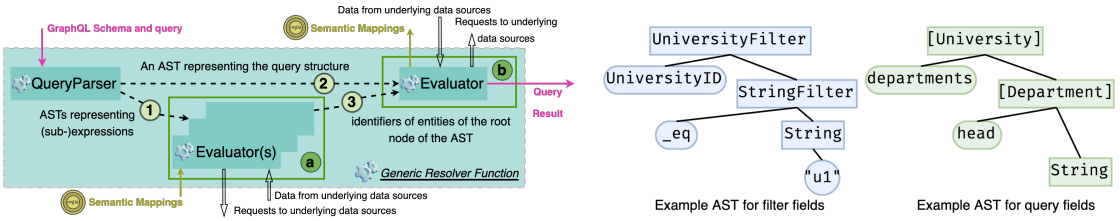
Our schema generator (as shown in Algorithm 1) first iterates over the concept names. For each concept (e.g., `University`), the concept name is used as the name of type to be generated in the GraphQL schema (`University`); the term concatenated with 'Filter' is used as the name of an input type to be generated (`UniversityFilter`); the term concatenated with 'List' is used as the name of a field of the `Query` type (`UniversityList`). 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`). In the next step, the algorithm iterates over GCIs. Taking such a GCI, `University` $\sqsubseteq \forall$ `departments.Department`, as an example, the algorithm generates field definitions `departments: [Department]` of the `University` type, and `departments: [DepartmentFilter]` of the `UniversityFilter` type. For a GCI `University` $\sqsubseteq =$ 1 `UniversityID.String`, the algorithm generates field definitions `UniversityID: String` of the `University` type, and `UniversityID: StringFilter` of the `UniversityFilter` type.

Algorithm 1: *Schema Generator*

Input : a set of concepts, C; a set of GCIs, G
Output: a GraphQL schema $\mathcal{S}$

```
for P ∈ C do
    extend S with an empty object type, P
    extend S with an empty input type, PFilter
    add field/argument declarations to the Query type
for t ∈ G do
    if t is of the form P ⊑ Q then
        extend S with an empty interface type, Q
        extend S with an input type, QFilter
        extend S with field/argument declarations to
          the Query type
        extend S with declaration that P implements Q
    else
        /* t is of the other forms */
        extend S with field declarations to P, PFilter
```

The generic resolver function includes technical components *QueryParser* and *Evaluator* as shown in Figure 3a. The *QueryParser* parses a query including a filter expression given as an input argument, and outputs the corresponding abstract syntax trees (ASTs) for the input argument and the query structure, respectively. Figure 3b shows example ASTs for a filter expression and a query structure according to the query example in Figure 1c. The *QueryParser* parses the query, converts a filter expression into a union of conjunctive expressions (arrow ①), and generates an AST for each conjunctive expression and an AST for the query structure (arrow ②). Then, the filter expressions (frame ⓐ) and the query fields (frame ⓑ) 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 satisfying the conjunctive (sub-)expression. After a call to an evaluator based on an AST, data representing the requested type, which contains identifier information, is returned. During evaluation of the query fields, the identifier information is an input in the call to the evaluator (arrow ③). Taking the query in



(a) Outlined generic resolver function.

(b) Example Abstract Syntax Trees.

Figure 3: Overview of the generic resolver function.

Figure 1c as an example, the requested type is `University` and data that can identify university instances returns. Such identifier information captured in semantic mappings, is used to construct the URIs for subjects where such subjects represent *University* instances.

3. Demonstration

We demonstrate OBG-gen in a real-world data integration scenario in the materials design domain and in a synthetic benchmark scenario, Linköping GraphQL Benchmark (LinGBM) [3]. The demonstration is shown in a public page,[3] with pointers to an introduction video, detailed evaluation results and live GraphQL servers for the two demonstration scenarios.

Materials Design Domain Demonstration. This demonstration focuses on a real-world scenario in the field of materials design to integrate data from two data sources following different data models. We will demonstrate that the GraphQL server, generated based on the Materials Design Ontology (MDO) [4, 5], can provide integrated access to data from heterogeneous data sources (i.e., requests data with a single GraphQL query without materializing the underlying data). The domain ontology used by this demonstration aims to improve the interoperability in the field for data integration. Therefore, the generated GraphQL schema plays as an integrated view of materials design data. We write 12 GraphQL queries in total among which 7 are with filtering conditions. Some of the queries are of domain interest written based on competency questions used for developing MDO. The other queries are written for testing the functionalities of the tool. One example query, as shown in Listing 1, is to get all the calculations pertaining to silicon-based materials with band gap property above 2.0.[4]

LinGBM Demonstration. LinGBM is a performance benchmark for GraphQL server implementations. It provides a scalable dataset regarding the *University* domain and specifies key technical challenges (e.g., relationship traversal) of GraphQL server implementations. In addition, it contains query templates covering different technical challenges. Therefore in this scenario, we focus on demonstrating: (1) the generability and applicability of our approach for data access in a different domain; (2) the current coverage of our approach in terms of key technical challenges (e.g., attribute retrieval, relationship traversal, searching and filtering). We use the GraphQL schema provided by LinGBM and manually define semantic mappings

[3] https://liusemweb.github.io/obg-gen/demo/

[4] This query is of domain interest, because semiconductor materials with band gaps above 2 electronvolts are referred to as wide-bandgap semiconductors. Such semiconductors are widely used in various electronic devices.

Listing 1: Example query to get calcuations of silicon-based materials with band gaps above 2.0.

```
{
  CalculationList(
     filter: {
        _and: [
           {
              hasOutputStructure: {
                 hasComposition: { ReducedFormula: { _like:  "Si" } } }
           },
           {
              hasOutputCalculatedProperty: {
                 _and: [ { PropertyName: { _eq: "Band Gap" } }, { numericalValue: { _gt: 2.0 } } ] }
           } ] }
     )
     {
        ID
        hasOutputCalculatedProperty {
           PropertyName
           numericalValue
        }
     }
}
```

to construct a GraphQL server. We select 7 query templates from LinGBM to create query instances. One query template is used to construct queries that request all the publications of which the titles contain a specific string (e.g., "`formalization`").

4. Conclusion

This paper has briefly introduced the OBG-gen, a prototype implementation for generating GraphQL servers. Using OBG-gen, GraphQL application developers can avoid constructing GraphQL servers from scratch. In the future, we will work on supporting more query features (e.g., order by) in the generic resolver function; follow the development of the GraphQL language and explore the possibility of formally generating new features based on ontologies.

References

[1] D. Calvanese, G. De Giacomo, Data Integration: A Logic-Based Perspective, AI magazine 26 (2005) 59–59. doi:10.1609/aimag.v26i1.1799.

[2] H. Li, Ontology-Driven Data Access and Data Integration with an Application in the Materials Design Domain, Ph.D. thesis, 2022. doi:10.3384/9789179292683.

[3] S. Cheng, 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, 2022. doi:10.1007/978-3-031-20891-1_16.

[4] H. Li, R. Armiento, P. Lambrix, An Ontology for the Materials Design Domain, in: The Semantic Web - ISWC 2020 - 19th International Semantic Web Conference, 2020. doi:10.1007/978-3-030-62466-8_14.

[5] P. Lambrix, R. Armiento, H. Li, O. Hartig, M. Abd Nikooie Pour, Y. Li, The materials design ontology, Semantic Web (2023). doi:10.3233/SW-233340.