GraphSPARQL: A GraphQL Interface for Linked Data

Kevin Angele
Department of Computer Science,
University of Innsbruck
Innsbruck, Tyrol, Austria
kevin.angele@uibk.ac.at
Onlim GmbH
Austria
kevin.angele@onlim.com

Manuel Meitinger
Department of Computer Science,
University of Innsbruck
Innsbruck, Tyrol, Austria
manuel.meitinger@student.uibk.ac.at

Marc Bußjäger
Department of Computer Science,
University of Innsbruck
Innsbruck, Tyrol, Austria
marc.bussjaeger@student.uibk.ac.at

Stephan Föhl
Department of Computer Science,
University of Innsbruck
Innsbruck, Tyrol, Austria
stephan.foehl@student.uibk.ac.at

Anna Fensel
Department of Computer Science,
University of Innsbruck
Innsbruck, Tyrol, Austria
anna.fensel@uibk.ac.at
Wageningen University & Research
Wageningen, The Netherlands
anna.fensel@wur.nl

ABSTRACT

In recent years, knowledge graphs have become widely adopted for storing and managing vast amounts of data, powering various applications. However, SPARQL as the query language for accessing those knowledge graphs has a steep learning curve and is too complex for many use cases. This paper presents GraphSPARQL, a middleware that allows accessing arbitrary SPARQL endpoints by using GraphQL, supporting the GraphQL operations *query* and *mutation*. GraphSPARQL abstracts the complexity of SPARQL without losing the ability to address classes and properties of distinct ontologies. Additionally, GraphSPARQL's extension to GraphQL allows using SPARQL filter operations to filter the data in queries. The evaluation showed that GraphSPARQL can compete with existing GraphQL to SPARQL solutions and outperforms them for deeply nested queries.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; • **Information systems** → **Data management systems**;

KEYWORDS

GraphSPARQL, GraphQL, Knowledge Graphs, SPARQL, Linked Data

SAC '22, April 25–29, 2022, Virtual Event,

ACM ISBN 978-1-4503-8713-2/22/04…$15.00
https://doi.org/10.1145/3477314.3507655

ACM Reference Format:
Kevin Angele, Manuel Meitinger, Marc Bußjäger, Stephan Föhl, and Anna Fensel. 2022. GraphSPARQL: A GraphQL Interface for Linked Data. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event, .* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3477314.3507655

1 INTRODUCTION

In 2012 Google announced their Knowledge Graph [13] for improving their search services. This announcement can be seen as the starting point for the hype around knowledge graphs, which are, in principle, "large semantic nets" [5]. Since then, the number of companies using knowledge graphs to store, maintain and link their data has grown steadily. Nowadays, many large companies, like Google, Facebook, Amazon, or Airbnb, rely on knowledge graphs. Besides, this trend is also represented by the evolution of The Linked Open Data Cloud[1] (LOD Cloud) and especially the large open knowledge graphs DBPedia [1] and Wikidata [17] as the core of the LOD Cloud.

For accessing and managing the data within knowledge graphs SPARQL[2], a W3C recommendation is provided. SPARQL is a very flexible query language that allows writing complex queries for retrieving or modifying the data. However, this flexibility of SPARQL has its downsides. Due to the lack of proper ontology descriptions or example queries of SPARQL endpoints, retrieving data from a knowledge graph is very time-consuming. Additionally, this lack of documentation results in a steep learning curve for beginners. Hence, efficiently working with SPARQL endpoints requires intimate knowledge about the data structure. Another downside is that developers are not familiar with SPARQL as query language and its triple-based output. Therefore, developers can not use the full potential of knowledge graphs. An idea that might appear is

[1] https://lod-cloud.net/
[2] https://www.w3.org/TR/rdf-sparql-query/

domain experts defining templates for SPARQL queries that the developers then use in their applications to retrieve the relevant data. This idea brings two significant issues. First, curating and reusing predefined SPARQL queries is sub-optimal and will probably result in hardwired queries in the application code. Second, the problem of handling SPARQL's triple-based output remains. Handling those results requires more efforts from the developers as they typically work with nested objects (document-based data structures) rather than triples (graph-oriented data structure) [11]. Another major drawback of SPARQL is its awareness. SPARQL is very well known in the research domain, but the majority of developers that are used to JSON and REST APIs still do not know SPARQL. The drawbacks mentioned above result in the challenge of providing a layer of abstraction that hides the complexity SPARQL brings while keeping the flexibility it allows (as much as possible).

In this paper, we introduce a layer of abstraction based on the well-known nested objects structure (i.e., JSON) familiar to developers to simplify the access to knowledge graphs. This abstraction layer allows users unfamiliar with SPARQL to browse the schema of the underlying data and access the knowledge stored within a knowledge graph. As the basis for the abstraction layer, a query language well adopted by developers called GraphQL[3] is used. GraphQL was developed by Facebook and open-sourced in 2015. The main benefits of GraphQL as a query language are its simplicity in usage and a large number of available tools for it [14]. Initially, GraphQL was invented as an alternative for REST APIs, allowing developers to specify exactly the data they need. The main aim was to reduce the amount of data sent between client and server to save bandwidth and mobile applications' data consumption. In recent years, GraphQL gained attraction in the Semantic Web community as a less complex query language[4]. GraphQL is already supported by some of the major graph database providers, like Ontotext GraphDB[5] or Stardog[6]. With GraphQL being a new technology for accessing knowledge graphs, not all graph databases that one might want to access (fully) support its functionality. In conclusion, a query language adopted from GraphQL is less expressive than SPARQL. Still, it allows realizing typical data retrieval tasks.

The previously mentioned layer of abstraction is implemented in a toolkit called GraphSPARQL. GraphSPARQL provides a GraphQL interface for arbitrary SPARQL endpoints. However, this abstraction of SPARQL comes with a reduced expressivity, e.g., GraphQL's nature of nested documents does not allow `SPARQL SELECT` queries. The structure of GraphQL queries is more familiar to the structure of `SPARQL CONSTRUCT` queries. To minimize this gap between the abstraction layer and SPARQL, GraphSPARQL allows custom extensions to GraphQL (see Section 4.2). The contribution of GraphSPARQL includes:

- Automatic generation[7] of a GraphQL schema based on given ontologies in RDF/OWL format. This schema is then extendible by custom extensions (e.g., additional types or fields that should be queryable). The generation process allows integrating various ontologies into a single GraphQL schema by prepending a prefix to type and field names. Such a prefixing mechanism allows querying data from different ontologies without violating the GraphQL specification (GraphQL does not foresee the usage of namespaces).
- GraphQL comes with many open source editors like GraphiQL, allowing one to browse the underlying data schema easily. Browsing the schema simplifies the process of defining queries to retrieve the relevant data. GraphSPARQL supports GraphiQL to define GraphQL queries for the underlying schema.
- GraphQL queries sent to GraphSPARQL are transformed into SPARQL queries and sent to the specified SPARQL endpoints. GraphSPARQL allows defining different SPARQL endpoints for different ontologies. Based on the used types and fields, the corresponding SPARQL endpoint is called to retrieve the results.
- Before returning the answer to the user, GraphSPARQL parses the SPARQL results and transforms them back into a GraphQL result. GraphQL has a nested objects structure that is familiar to developers.

GraphSPARQL is implemented in `C#`[8] and utilizes existing GraphQL and SPARQL libraries to transform GraphQL queries to SPARQL queries and SPARQL results to GraphQL results.

The paper is structured as follows. In Section 2, we present proprietary and open source solutions providing a GraphQL interface for SPARQL and alternative approaches for abstracting the complexity of SPARQL for accessing a knowledge graph. Afterward, Section 3 provides an analysis of the requirements and an open-source tool called HyperGraphQL. Implementation details such as the architecture of GraphSPARQL can be found in Section 4. Section 5 presents the benchmark results. Finally, we conclude our paper in Section 6.

2 RELATED WORK

Different approaches for simplifying the access to and the modification of knowledge graphs have been developed. One type of approach are so-called domain-specific languages like LDFlex [16]. LDFlex is specially designed for front-end web developers to fit their workflow. As holds for all the approaches we present in this section, the main goal is to abstract the complexity of SPARQL. Usual problems solved by front-end web developers are not very complex, so a simpler language is sufficient. LDFlex is comparable with jQuery[9] but for RDF.

Another type of approach is REST APIs built on top of knowledge graphs that allow accessing and modifying the underlying data, like RAMOSE [4] or OBA [6]. Based on a given ontology OBA, for example, automatically creates a REST API that allows to get and manipulate data. OBA relies on the OpenAPI-Specification[10] and provides a server implementation based on the created specification. This approach requires direct access to the ontology and especially the data. Therefore, it is limited to the RDF data provider.

[3] https://graphql.org/
[4] In the future GraphQL will become even more important, see https://www.bbntimes.com/technology/why-graphql-will-rewrite-the-semantic-web
[5] https://www.ontotext.com/products/graphdb/
[6] https://www.stardog.com/
[7] The manual definition of a GraphQL schema is possible as well.
[8] https://docs.microsoft.com/en-us/dotnet/csharp/
[9] https://jquery.com
[10] http://spec.openapis.org/oas/v3.0.3

Another approach in this direction is [11] *Easy Web API development with SPARQL transformer* defining the queries and the expected output in a JSON format. Those files are processed by the SPARQL Transformer[11], stored in a GitHub repository, and with the help of grlc[12] a RESTful API is created. Besides, a cloud platform for sharing and reusing SPARQL queries is BASIL [3]. BASIL automatically generates Web APIs from SPARQL queries easily embeddable into applications. Work in pharmacology databases is the Open PHACTS Discovery Platform leveraging Linked Data to provide integrated access to those databases. [7] extend the Open PHACTS platform with APIs to facilitate data integration.

A general approach that aims to simplify the usage of RDF data is EasierRDF[13]. However, the development in that direction appears to be relatively static (the last commit was two years ago). EasierRDF aims to simplify tools, standards, and documentation to make RDF usable by a broader audience.

In recent years GraphQL as an abstraction of SPARQL has evolved. Since then, a couple of commercial and open-source solutions have appeared. Those solutions use GraphQL to access RDF data directly or transform GraphQL queries to SPARQL queries to access arbitrary SPARQL endpoints. A **commercial solution** in that regard is Stardog[14]. Stardog offers a graph database called the "Enterprise Knowledge Graph platform". The main interface to the stored data is given via SPARQL, and their extension of SPARQL is called Pathfinder[15]. Pathfinder is optimized for finding and retrieving arbitrarily complex paths from within the stored knowledge graph. Their platform supports GraphQL as of version Stardog 5.1[16]. Stardog also allows users to provide a GraphQL schema to translate RDF and GraphQL values and validate the queries. Besides, Stardog allows schemaless GraphQL queries to access the underlying data. However, this requires intimate knowledge of the underlying data structure. Another relevant solution is "TopBraid Enterprise Data Governance (EDG)"[17] by TopQuadrant. TopBraid EDG uses RDF to store data and allows users to define data models using RDF Schema [2], SHACL [10] and OWL [18]. Furthermore, a GraphQL schema can be defined, and GraphQL can be used to query the database[18]. TopBraid EDG also allows users to nest SPARQL queries within GraphQL queries, making the queries overall more explicit. A further commercial solution is "Ontotext Platform"[19] by Ontotext. Ontotext Platform offers users a GraphQL interface to query an underlying graph database called GraphDB[20]. As with the solutions above, the underlying GraphDB can also be queried using SPARQL. Those examples show that major graph database solutions already adopt GraphQL.

An **open-source solution** is GraphQL-LD by Taelman et al. [14]. GraphQL-LD is the first academic open-source solution that translates GraphQL queries to SPARQL queries and SPARQL results to GraphQL results. In contrast to the commercial approaches mentioned above, GraphQL-LD can be used for arbitrary graph databases that offer a SPARQL endpoint. Instead of using GraphQL schema, GraphQL-LD uses a JSON-LD [9] context to create a mapping between the RDF terms of the database and GraphQL. Additional features that GraphQL-LD supports are filtering and ordering through directives [15]. Another open-source solution is HyperGraphQL[21]. Similar to GraphQL-LD, HyperGraphQL offers a GraphQL interface for arbitrary SPARQL endpoints. In contrast to GraphQL-LD, HyperGraphQL uses GraphQL schema (instead of JSON-LD) to add semantic meaning to GraphQL. Initially, GraphSPARQL was meant to be an add-on to HyperGraphQL, as HyperGraphQL overlapped most with our project's goals. However, after a more in-depth analysis, we decided against writing an add-on and for creating our own implementation. This analysis of HyperGraphQL and the reasons behind our decision can be found in Section 3.

We aim to provide an interface for arbitrary SPARQL endpoints that allow connecting to multiple endpoints simultaneously. Therefore, the proprietary solutions are not usable for us, as they are tied to a specific database. The REST API approach needs direct access to the data and is also not an option for us. Besides, EasierRDF is still in the design phase and not yet very concrete. As a result, we will focus on the existing open-source solutions.

3 REQUIREMENT ANALYSIS

This paper presents a GraphQL to SPARQL solution to query data from knowledge graphs and modify data within them. For benefiting from existing tools for GraphQL, it is essential to stay as close to the GraphQL specification as possible. For mapping GraphQL to SPARQL, the platform or user has to provide an additional schema that semantically annotates the queries' fields. Providing such a schema is typically feasible. However, when trying to integrate federated queries into a GraphQL interface, the different sources' schema must be compatible, e.g., not introduce ambiguities. This implies that fields with the same name used for various sources must be distinguishable within the GraphQL queries. A conceptual limit of GraphQL's capabilities is given by the fact that nested objects represent trees. Thus, the results of GraphQL queries can only be trees (in principle restricted *SPARQL CONSTRUCT* queries). In contrast, SPARQL can return single values (*ASK* or *SELECT*) or arbitrary graphs as a result, which includes trees and is therefore strictly more powerful.

Data within a knowledge graph is typically structured in hierarchies, and values of properties can be a disjunction of types. Therefore, GraphQL `enum`, `interface` and `union` need to be supported.

Based on the idea of simplifying accessing and managing knowledge graphs while staying as close to the GraphQL specification as possible, the following requirements can be derived:

- connection to arbitrary SPARQL endpoints
- federated queries using distinct ontologies
- query and modify data
- support for GraphQL `enum`, `interface`, and `union`
- advanced filtering options (using SPARQL filters)

[11] https://d2klab.github.io/sparql-transformer/
[12] http://grlc.io
[13] https://github.com/w3c/EasierRDF
[14] https://www.stardog.com/
[15] https://www.stardog.com/blog/a-path-of-our-own/
[16] https://www.stardog.com/blog/graphql-and-paths/
[17] https://www.topquadrant.com/products/topbraid-enterprise-data-governance/
[18] https://www.topquadrant.com/technology/graphql/
[19] https://www.ontotext.com/products/ontotext-platform/
[20] http://platform.ontotext.com/index.html
[21] https://www.hypergraphql.org/

- mark fields as required (they must be provided by an instance to be returned)
- extended set of scalar types (e.g., `DateTime` and `LanguageString`)

Based on those requirements we analyzed existing open-source solutions. One of the available open-source solutions that aligns mostly with our project requirements is HyperGraphQL. However, the following functionality is not given by HyperGraphQL:

(1) support for **enum, interface** and **union**
(2) **mutation** and **advanced filtering** operations
(3) support for required fields in queries

Therefore, our initial intention was to implement an add-on to HyperGraphQL. After analyzing HyperGraphQL in more detail, we encountered a few more issues that led to the realization that an add-on is more work than a complete re-write. The first issue we encountered is that the latest release (version 2.0.0 on June 2021) upgrades libraries and the java version without providing additional functionality or improving the existing functionality. The prior release was version 1.0.1, published in October 2018. Thus, the development of HyperGraphQL seems relatively static. Combined with the fact that close to no documentation and no comments are available for the code, getting familiar with the project becomes significantly harder. This issue is, of course, not a roadblock. However, it drastically slows down the implementation process.

As for technical issues, HyperGraphQL has to tackle the inherent limitations of using a GraphQL interface for a SPARQL interface. These limitations lead to inherent schema limitations in HyperGraphQL, namely that all fields have to be global. Global means that when looking at a HyperGraphQL schema, all types and fields need to be declared in a global section before fields can be specified with their corresponding properties[22]. Furthermore, HyperGraphQL does not support GraphQL abstraction semantics, meaning that only concrete types can be used. The final noteworthy issue we encountered when analyzing HyperGraphQL is that no typed scalar handling is given. This means that if we, e.g., query for a time or duration, the return value is always a string instead of a date-time object or a numeric data type.

Note that GraphQL-LD has similar issues as HyperGraphQL. Furthermore, we did not consider extending GraphQL-LD since it requires a JSON-LD schema that could limit the achievable functionality and introduce another inhibition threshold for developers that are not familiar with working with linked data.

4 GRAPHSPARQL

As stated above, HyperGraphQL has been an inspiration for GraphSPARQL, in the sense that both can be used as middleware between GraphQL queries and SPARQL endpoints. Furthermore, both solutions are accompanied by a GraphiQL[23] interface (see Figure 1), which simplifies the submission of GraphQL queries and visualizes the returned results.

In the following, we first elaborate more on the architecture of GraphSPARQL (Section 4.1), starting with an overview and then delving deeper into how queries and results are translated. Section 4.2 concludes the theoretical discussion of GraphSPARQL by

[22] see the Schema section on https://www.hypergraphql.org/documentation/
[23] https://www.electronjs.org/apps/graphiql

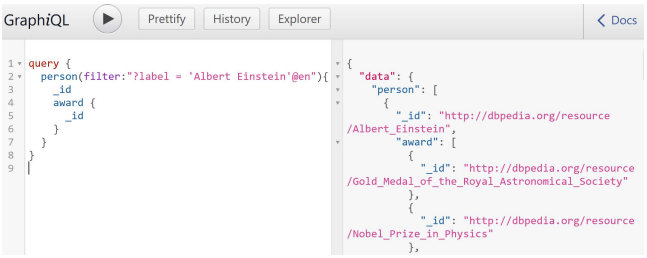

Figure 1: Screenshot of the GraphiQL user interface. The left side presents a query and the result is presented on the right.

elaborating on the generation of the enriched GraphQL schema that is used to overcome the functionality limitations of GraphQL (see Section 3). Afterward, we will introduce extensions to the GraphQL query syntax for the advanced filtering options (Section 4.3). Finally, the process of generating SPARQL queries from GraphQL queries is described (Section 4.4).

4.1 GraphSPARQL Architecture

GraphSPARQL's general architecture consists of four basic building blocks:

(1) an *Enriched GraphQL Schema* that is used to provide semantic meaning for the GraphQL queries
(2) an *RDF Harvester* that generates said schema
(3) a *Query Processor* that utilizes the *dotNetRDF*[24] library to send SPARQL queries to and receive SPARQL results from connected RDF databases
(4) a *Parser & Formatter* that utilizes the *GraphQL .Net*[25] library to translate the GraphQL queries for and the SPARQL results from the *Query Processor*

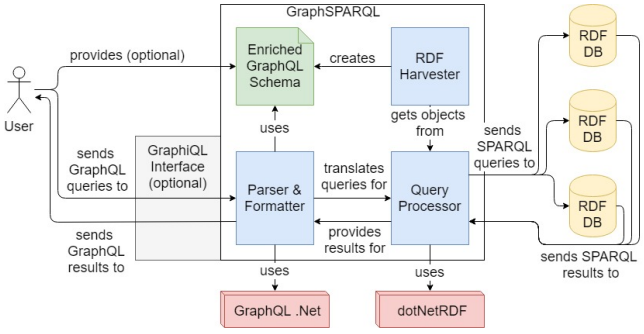

Figure 2: GraphSPARQL architecture overview.

An overview of this architecture can be found in Figure 2. Starting from a user's perspective, GraphSPARQL is set up by connecting it to one or more SPARQL endpoints via a JSON config file. Besides, at least one GraphQL schema source has to be defined in the said file as well. Currently, the following sources are supported:

[24] https://www.dotnetrdf.org/api/html/N_VDS_RDF.htm
[25] https://graphql-dotnet.github.io/docs/getting-started/introduction

(1) RDFS/OWL schema: This option uses the *RDF Harvester* to generate the GraphQL schema. Various flags can be set to govern what exactly is being harvested (e.g., what namespaces, datatypes, unused types).
(2) *Enriched GraphQL Schema* (see Section 4.2): Here a GraphQL schema is used as input. However, since GraphQL does not support any SPARQL hints out-of-the-box (like what endpoint to use or how to map a GraphQL field to a SPARQL/RDF property), additional *directives* have to be supplied.
(3) JSON: The last option allows to provide GraphQL schema(s) in the internal JSON format. The library also supports exporting compiled schemas into that format. This option is the fastest and supports embedding a schema into the config file rather than loading it from an external resource.

If multiple schemas are defined, they can also reference each other, so it is possible to harvest some GraphQL types from an OWL schema and then define additional properties on those types through another (enriched) GraphQL or JSON schema.

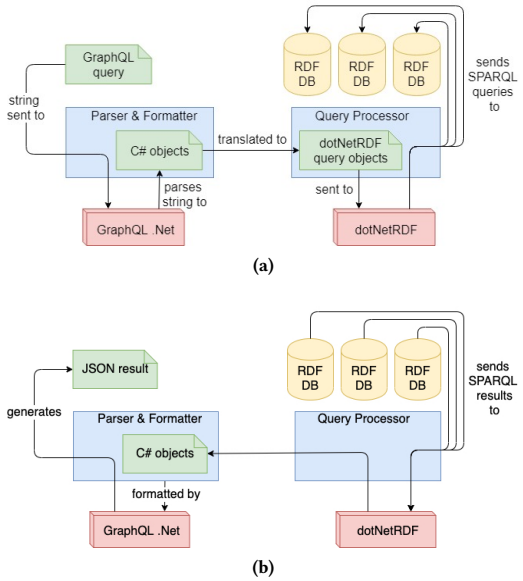

Figure 3: GraphSPARQL architecture close-up. (a) Issuing a GraphQL query. (b) Receiving results.

With the endpoint and schema configuration available, the user can send their GraphQL queries (see Figure 3a for a close-up) to the *Parser*, optionally using the *GraphiQL* interface in the process. Utilizing the *GraphQL .Net* library as well as C#-objects generated from the specified schema, the *Parser* translates and collects queries which are then given in batches to the *Query Processor*, which in turn uses the *dotNetRDF* library to send SPARQL queries to the connected RDF databases.

Considering the returned results (see Figure 3b), the *Query Processor* receives SPARQL results from the queried RDF databases. The *dotNetRDF* library is used to lift all the different return formats. These results are then fed back to the generated C# schema objects that either use the formatting capabilities provided by the *GraphQL .Net* library or our custom scalar formatters to produce the JSON results. Finally, the JSON data is then again optionally displayed by the *GraphiQL* interface.

Listing 1: Example of the extended GraphQL schema. An `Adult` is a `Person` that drives a `Car`, a `Child` is a `Person` that is not an `Adult`, and a `Car` is a union of different car brands. The `name` field is inherited by `Adult` and `Child`. Each car has to define the `type` field itself, since `Car` is a union and not an interface.

```
interface Person { name: String! }

enum FuelType @datatype("http://www.w3.org
    /1999/02/22-rdf-syntax-ns#langString") {
    GASOLINE @value(literal:"'Benzin'@de"),
    ELECTRIC @value(literal:"'Strom'@de"),
}

@context(datasource:"CarDB")
type Ford implements Car { type: FuelType }

@context(datasource:"CarDB")
type Tesla implements Car { type: FuelType }

union Car = Ford | Tesla

type Adult implements Person {
    drives: Car @predicate(datasource:"CarDB", graph:
        "http://example.com/cars/peoplegraph")
}

type Child implements Person { parents: [Adult!] }

type Query {
    persons(ids:[String!], filter:String, offset:Int,
            limit:Int, require:[String!]): [Person!]
}
```

4.2 Enriched GraphQL Schema

As alluded to in Section 3, a standard GraphQL schema is not enough to provide sufficient information on how to map GraphQL types and fields to RDF classes and properties. Furthermore, as we want to support advanced filtering, mandatory fields, and other query flags that match SPARQL behavior, we have to store additional meta-data within the schema (see Listing 1).

The syntax of GraphSPARQL's enriched schema is identical to GraphQL. The main differences are that the enriched schema allows for inheriting fields from interfaces, so one does not have to define the field in every implementing type, and additional *directives*, like `@class` for setting the `rdf:type` IRI of a GraphQL type or `@value` for providing the SPARQL value of an `enum`.

To reduce the verbosity of the schema, these directives are also inherited, so setting the namespace on a GraphQL type by using the `@context` directive ensures that this namespace is also used to generate the IRIs of the fields of this type. One can even specify a default SPARQL data source and a default namespace in the configuration file.

4.3 GraphQL Extended Query Syntax

Finally, the query expressiveness gap has to be addressed. In Listing 1 the `persons` field of the `Query` type defines special parameters. These parameters are automatically added to all query fields. While some are self-explanatory like `offset` and `limit`, which behave

Listing 2: Generic GraphQL query of depth 3.

```
query {
 {
  graduateStudent {
    advisor (filter: "bound(?name) &&
                      ?name='AssistantProfessor3'") {
      name
    }
  }
 }
}
```

similarly to their SPARQL counterparts, others are used to filter results in the following way:

(1) `id`/`ids`: Depending on whether the field is an array or not, this parameter restricts the result to either one or more IRIs. For top-level queries, the filter is applied to the subject. For others on the object of triples.
(2) `filter`: This parameter allows an arbitrary string to be passed to the SPARQL endpoint as `FILTER` expression. GraphQL fields can be used as well by prepending a ? to their names. See Figure 1 for an example. It is even possible to go down the object hierarchy, e.g., specifying `filter:"?child_name = 'Smith'"` on a parent object would query all parents that have a child with the name Smith.
(3) `require`: Syntactic sugar for specifying `BOUND` filters on each of the given field names.

Since GraphSPARQL also supports creating and updating objects and fields, there are even more parameters available on `Mutation` fields. A sample can be found in the project's README[26].

4.4 GraphSPARQL SPARQL Query Generation

In this section, the transformation from an incoming GraphQL query towards the outgoing SPARQL queries is described. As mentioned before, independent of the top-level elements used in a GraphQL query, one SPARQL query per level is constructed and sent to the SPARQL endpoint(s). For example, a query consisting of three levels results in three SPARQL queries. Currently, this split is required by the underlying GraphQL library used in GraphSPARQL. The underlying GraphQL library needs the type information to support unions.

In the following, a GraphQL query including a SPARQL filter (see Listing 2) is transformed into the corresponding SPARQL queries.

Based on the given GraphQL query, first, a SPARQL query for the top-level elements is created. There is only one top-level element in this example. Therefore the SPARQL query retrieves only instances of type `GraduateStudent`, as specified in the GraphQL query. To retrieve the relevant information, GraphSPARQL produces `SPARQL CONSTRUCT` queries. The resulting SPARQL query for retrieving the instances of the given type is shown in Listing 3.

Listing 3: SPARQL query concerning top-level elements.

```
CONSTRUCT { ?__s0 :p0 ?__o0 . } WHERE
{
    ?__o0 rdf:type ?__s0 .
    FILTER(?__s0 = :GraduateStudent)
}
```

[26]https://github.com/Meitinger/GraphSPARQL/blob/main/README.md

The result of this initial SPARQL query is a list of URIs identifying instances of type `GraduateStudent`. Those URIs are used for the second query to retrieve the values that are linked via the given property `:advisor` in the GraphQL query. Besides, the SPARQL filter specified on this property is propagated to the SPARQL query by replacing the variable name with a constructed variable name. This replacement is necessary to simplify the internal processing of this variable. Other than that, the given filter expression is forwarded without any change.

Listing 4: SPARQL query concerning the second level.

```
CONSTRUCT { ?__s0 :p0 ?__o0 . } WHERE
{{   { VALUES ( ?__s0 )
       {( <http://example.org/GraduateStudent0> )}
    }
    UNION
    {
      ?__s0 :advisor ?__o0 .
      OPTIONAL { ?__o0 :name ?__v0_0 . }
      FILTER(BOUND(?__v0_0) && (?__v0_0 = "
          AssistantProfessor3"))
    }
  }}
```

The result of this query is again used to retrieve the values linked via the next embedded property `name` (see Listing 5).

Listing 5: SPARQL query concerning the third level.

```
CONSTRUCT { ?__s0 :p0 ?__o0 . } WHERE
{{ VALUES ( ?__s0 )
    {( <http://example.org/AssistantProfessor3> )}
    }
    UNION
    { ?__s0 :name ?__o0 . }
}
```

Listing 5 shows the last query issued for the given GraphQL query in Listing 2. Afterward, the result is composed into a GraphQL result and sent back to the user.

5 EVALUATION

GraphSPARQL is evaluated against the existing open-source solution HyperGraphQL and Stardog. This setup allows a comparison between commercial, native support for GraphQL and an open-source solution. In this section, we first present our benchmark setup and then the realization of the benchmark.

5.1 Setup

As graph database for this benchmark, we used *Stardog*. It is used to store the data, and the SPARQL endpoint is used for benchmarking GraphSPARQL and HyperGraphQL. Additionally, the GraphQL queries were executed directly on the GraphQL endpoint of *Stardog*. Overall, this results in a comparison of the GraphQL performance for GraphSPARQL, HyperGraphQL, and Stardog. We installed Stardog on a test server[27] directly, without using docker containers.

The dataset used for the benchmark is from *Salzburgerland*[28]. It consists of approximately 31 k triples representing linked data

[27]https://www.hetzner.com/dedicated-rootserver/ex62-nvme - Processor: Intel® Core™ i9-9900K Octa-Core, 8 Cores / 16 Threads, 3.60 GHz Base Frequency, 5.00 GHz Max Turbo Frequency - RAM : 64 GB DDR4 - Hard Drive: 2 x 1 TB NVMe SSD - Operating system: Debian GNU / Linux 10
[28]It is available together with the GraphSPARQL code on GitHub - https://github.com/Meitinger/GraphSPARQL

on tourism in Salzburg, Austria. Choosing such a small dataset has the advantage of facilitating manual testing and debugging during development. Furthermore, as HyperGraphQL forces the user to specify a schema file, using a smaller dataset simplifies the creation of such a file.

5.2 Benchmarks

After setting up Stardog with the *Salzburgerland* dataset, we benchmarked GraphSPARQL by comparing it to the GraphQL interface of Stardog and HyperGraphQL. GraphSPARQL and HyperGraphQL are thereby connected to the SPARQL endpoint of the Stardog graph database.

As all three interfaces use a slightly different syntax, we manually formulate sets of GraphQL queries and write parsing scripts to translate them for the different interfaces. This ensures that all tools are benchmarked with equivalent queries. Those queries are designed to work with all three tools equally and are kept simple to produce representative results. As we want to showcase that GraphSPARQL optimizes queries that include nesting, we group queries of different nesting depths. An example for a nested query using the *schema:knows* field can be found in Listings 6.

Listing 6: Exemplary query of nesting depth 2.

```
query {
  person{
    dctTitle
    knows {
      dctTitle
      knows {
        dctTitle
      }
    }
  }
}
```

To avoid variable network latency, we perform all measurements locally on the test server mentioned above, submitting the queries via a *bash* script using the *curl* function in the process. The *bash* scripts submit each of the specified queries one after another, measuring the time between submitting the queries and receiving the corresponding results. As the generation of statistically relevant data is desired, the execution times of all queries within one set are summed up, and the experiment is repeated 25 times.

As the GraphQL interfaces have to perform setup operations utilizing the provided schema files, the initial query should have a longer execution time than consecutive queries. Furthermore, resulting from repeating the experiments, repeated access to the same values might lead to caching. To reduce these effects, we restart the GraphQL interfaces every five rounds. Thus, we can statistically group the corresponding round's execution times using the arithmetic mean and standard deviation, leading to the results displayed in Figure 4. In other words, the data point of the i-th round corresponds to the mean of the j-th experiment with $i \equiv j \bmod 5$. The displayed error bars correspond to the respective standard deviation. Note that the Stardog GraphQL interface's initialization time is not easily separable from the graph database's initialization time. Thus, as we did not observe significant deviations between the initial and successive GraphQL queries to the Stardog interface, we only restart the GraphSPARQL and HyperGraphQL interfaces after every five rounds.

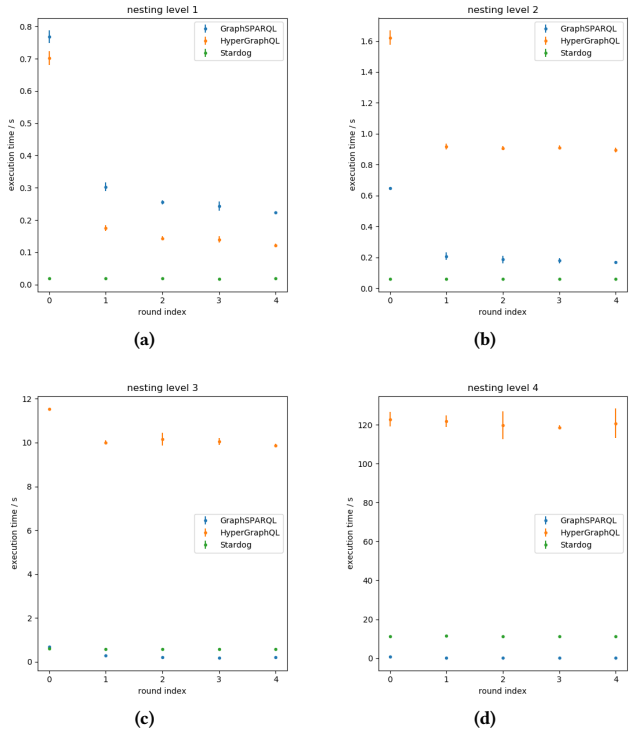

Figure 4: Benchmarking results grouped by nesting depths/levels and rounds as detailed in section 5.2.

The data displayed in Figure 4a-d shows the experimental results from queries with the corresponding nesting depth. As can be seen, in Figures 4a-c, HyperGraphQL, and GraphSPARQL have a comparative initialization time of about half a second. For the nesting depth 1 queries (see Figure 4a), we observe a declining trend of the execution time for HyperGraphQL and GraphSPARQL. We assume that this results from some caching by Stardog's SPARQL endpoint and/or the GraphQL interfaces.

As visible in Figures 4a-b, Stardog consistently provides the fastest GraphQL interface for queries of nesting depth 1 and 2. This might be a result of some internal optimization between Stardog's GraphQL interface and graph database. For queries of nesting depth 1 (see Figure 4a), the HyperGraphQL interface is about 0.1 s faster than GraphSPARQL. However, as shown in Figures 4b-d, GraphSPARQL significantly outperforms HyperGraphQL when increasing the nesting depth. With nesting depth 3 (see Figure 4c), GraphSPARQL outperforms Stardog's GraphQL interface (omitting the initial/round 0 value of GraphSPARQL). This trend continues for a nesting depth 4 (see Figure 4d) where all GraphSPARQL data points show the lowest execution time.

To better visualize the trend, we compare the execution times for their nesting depth in Figure 5. As the effect of caching has been smaller than anticipated, the execution times of rounds 1 to 4 are grouped using the mean and standard deviation. By plotting the execution time on a logarithmic scale, it is apparent that Stardog and HyperGraphQL show an exponential trend when increasing the

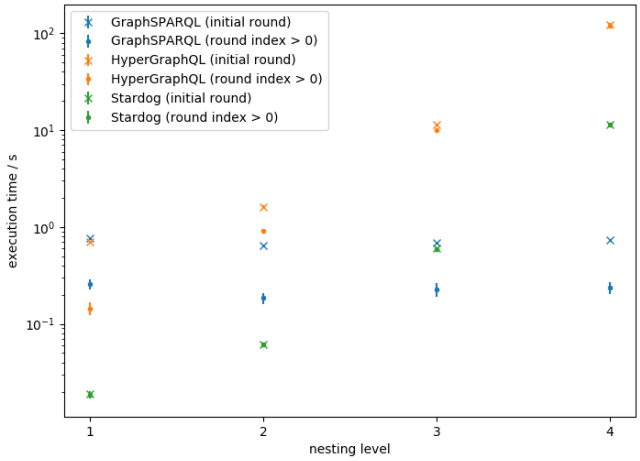

Figure 5: Execution time of GraphQL interfaces for increasing nesting depth/level.

nesting depth. GraphSPARQL, on the other hand, has an execution time that is constant within one standard deviation of the respective measurements. Less apparent due to the vertical axis's logarithmic scaling, the initial round 0 of GraphSPARQL and HyperGraphQL is constantly offset from their successive rounds for all nesting depths.

6 CONCLUSION

The wide adoption of knowledge graphs with SPARQL as flexible but complex query language comes with barriers for developers unfamiliar with SPARQL and its triple-based output. A layer of abstraction based on a nested object structure was presented for developers not familiar with the semantic web stack to realize typical data retrieval tasks. We introduced GraphSPARQL to implement this abstraction layer that simplifies access to knowledge graphs by using GraphQL, a query language well adopted by developers. GraphSPARQL supports accessing (corresponds to GraphQL *queries*) and manipulating (corresponds to GraphQL *mutations*) knowledge graphs. Bridging the gap between GraphQL as a less expressive query language and SPARQL, GraphQL provides an enriched GraphQL schema and advanced filtering options. Those advanced filtering options allow using more complex SPARQL filters if needed. As shown by the benchmarks, GraphSPARQL is comparable to existing solutions (e.g., HyperGraphQL) and outperforms those and native GraphQL implementations (like Stardog) when using deeply nested queries.

GraphSPARQL already provides a vast toolkit for accessing and managing knowledge graphs. In the future, we will extend the automatic GraphQL schema generation with the support for SHACL shapes. In addition to generating a GraphQL schema from ontologies, existing SHACL shapes are then usable. An existing benchmark consisting of 58 SHACL shapes [12] can be applied for evaluating the use of SHACL shapes for building the GraphQL schema. Additionally, we identified some limitations in the SPARQL query generation process. Therefore, we plan to replace the underlying GraphQL library to gain more flexibility in generating SPARQL queries. Besides, the performance of the GraphQL schema introspection will be improved to be used for large schemas. Currently, introspection is disabled, and an alternative way for inspecting the schema is provided (*_fields* query property). Finally, we plan to perform a much more extensive benchmark using the LUBM [8] benchmark datasets and queries.

ACKNOWLEDGEMENT

This work results from the Semantic Web course at the University of Innsbruck (WS 2020/2021) and has been supported by the project WordLiftNG within the Eureka, Eurostars Programme (grant agreement number 877857 with the Austrian Research Promotion Agency (FFG)). Umutcan Şimşek and Dieter Fensel (both from the University of Innsbruck) supported initial research in that direction, funded by the mindLab (https://mindlab.ai) project.

REFERENCES

[1] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *The semantic web*. Springer, 722–735.
[2] Dan Brickley, R.V. Guha, and Brian McBride. 2014. RDF Schema 1.1. World Wide Web Consortium (W3C). https://www.w3.org/TR/rdf-schema/
[3] Enrico Daga, Luca Panziera, and Carlos Pedrinaci. 2015. Basil: A cloud platform for sharing and reusing SPARQL queries as Web APIs. In *CEUR Workshop Proceedings*, Vol. 1486.
[4] Marilena Daquino, Ivan Heibi, Silvio Peroni, and David Shotton. 2021. Creating RESTful APIs over SPARQL endpoints using RAMOSE. *Semantic Web* 1 (2021), 1–19. https://doi.org/10.3233/sw-210439 arXiv:2007.16079
[5] Dieter Fensel, Umutcan Şimşek, Kevin Angele, Elwin Huaman, Elias Kärle, Oleksandra Panasiuk, Ioan Toma, Jürgen Umbrich, and Alexander Wahler. 2020. *Knowledge Graphs: Methodology, Tools and Selected Use Cases.* Springer Nature.
[6] Daniel Garijo and Maximiliano Osorio. 2020. OBA: An Ontology-Based Framework for Creating REST APIs for Knowledge Graphs. In *International Semantic Web Conference*. Springer, 48–64.
[7] Paul Groth, Antonis Loizou, Alasdair JG Gray, Carole Goble, Lee Harland, and Steve Pettifer. 2014. API-centric linked data integration: The open PHACTS discovery platform case study. *Journal of web semantics* 29 (2014), 12–18.
[8] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics* 3, 2-3 (2005), 158–182.
[9] Gregg Kellogg. 2020. *JSON-LD 1.1.* W3C Recommendation. W3C. https://www.w3.org/TR/json-ld11/
[10] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes Constraint Language (SHACL). World Wide Web Consortium (W3C). https://www.w3.org/TR/shacl/
[11] Pasquale Lisena, Albert Meroño-Peñuela, Tobias Kuhn, and Raphaël Troncy. 2019. Easy web API development with SPARQL transformer. In *International Semantic Web Conference*. Springer, 454–470.
[12] Robert Schaffenrath, Daniel Proksch, Markus Kopp, Iacopo Albasini, Oleksandra Panasiuk, and Anna Fensel. 2020. Benchmark for Performance Evaluation of SHACL Implementations in Graph Databases. In *International Joint Conference on Rules and Reasoning*. Springer, 82–96.
[13] Amit Singhal. 2012. Introducing the Knowledge Graph: things, not strings. https://blog.google/products/search/introducing-knowledge-graph-things-not/
[14] Ruben Taelman, Miel Vander Sande, and Ruben Verborgh. 2018. GraphQL-LD: linked data querying with GraphQL. In *International Semantic Web Conference*. 1–4.
[15] Ruben Taelman, Miel Vander Sandier, and Ruben Verborgh. 2019. Bridges between GraphQL and RDF. In *W3C Workshop on Web Standardization for Graph Data. W3C*.
[16] Ruben Verborgh and Ruben Taelman. 2020. LDflex: A Read/Write Linked Data Abstraction for Front-End Web Developers. In *International Semantic Web Conference*. Springer, 193–211.
[17] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85.
[18] World Wide Web Consortium et al. 2012. OWL 2 web ontology language document overview. (2012). https://www.w3.org/TR/2012/REC-owl2-overview-20121211/