Automatic Bootstrapping of GraphQL Endpoints for RDF Triple Stores

✉ Lars Gleim[1], Tim Holzheim[1], István Koren[1], and Stefan Decker[1,2]
{gleim@dbis,tim.holzheim@,koren@dbis,decker@dbis}.rwth-aachen.de

[1] Databases and Information Systems, RWTH Aachen University, Germany
[2] Fraunhofer FIT, Sankt Augustin, Germany

Abstract. GraphQL is a query language for graph-structured Web APIs, increasingly popular among Web developers and recently explored as an alternative query language for Linked Data and its underlying RDF data model. However, to date, the deployment of available GraphQL processors for RDF data requires users to have intricate knowledge of Semantic Web technologies, such as SPARQL and SHACL, as well as the schema of the underlying RDF data.We present Ultra-GraphQL (*UGQL*), an open source tool enabling the automatic bootstrapping of GraphQL endpoints for existing RDF triple stores, based on an adaptable SPARQL schema extraction, mapping and query translation approach. By automatically generating CRUD mutations for each object type, UGQL further enables write access to RDF data. UGQL thus allows developers with limited or no knowledge of Semantic Web technologies to read and write RDF data using plain GraphQL, eliminating dependencies on third-party schema definitions. By effectively lowering the entry barrier for working with Linked Data, it has the potential to be a ground-breaker for Semantic Web technologies.

Keywords: Linked Data Querying · Web of Data · Web Engineering

1 Introduction

As the overall amount of social, machine and transactional data generated and collected on the Web continues to grow rapidly, Linked Data and Semantic Web technologies play an increasingly important role in managing data in practical applications. For instance, the Industrial Internet of Things [1] promises to improve interoperability and easy integration of data silos [2] by providing semantically adequate and context-aware data. However, the Resource Description Framework (RDF) [3] data model of Linked Data and SPARQL [4] are unfamiliar to most developers [5]. The descriptive structure of SPARQL queries, distinct from more popular query languages such as SQL, as well as the respective query results, are criticized for containing unnecessary metadata and carrying duplicate information [6], often requiring additional parsing or transformation steps to allow for their usage in Web applications [7]. Additionally, in the context of mobile application development, the structure and size of the result should be as compact and small as possible, since network bandwidth and computational power are limited [8]. Overall, these challenges limit the practical adoption of Linked Data queried through SPARQL, as depicted in Figure 1a).

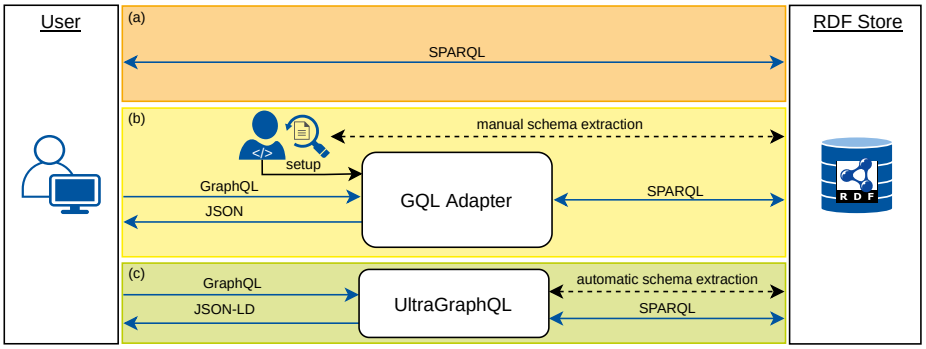


Fig. 1: Three modes of Linked Data querying: (a) direct SPARQL queries, (b) GraphQL queries via a GraphQL adapter and (c) GraphQL queries via UGQL

In contrast, GraphQL [9] has been specifically designed for mobile and Web applications. It features a tree-like structure, allowing for the traversal of the underlying graph. The syntax is mirrored by the result, reducing additional transformation, data redundancy and size compared to SPARQL [10]. GraphQL endpoints typically provide a schema introspection endpoint, supporting query auto-completion, based on a schema grammar with object type and field (property) definitions. An in-depth introduction to GraphQL can be found in [11]. Notably, popular GraphQL implementations are not directly suited for linked data applications, since they typically implement neither RDF compatible data serializations, nor global identifiers by default, rendering them incompatible with fundamental Semantic Web principles.

Recently, several publications have explored combining GraphQL with RDF compatible serializations [12,13,14,15,16]. However, those approaches require significant knowledge of Semantic Web technologies for their deployment and usage, or involve manual setup steps that limit their usage with third-party data sources, as illustrated in Figure 1b). To address this issue, we propose the fully automatic bootstrapping, i.e. setup and configuration, of GraphQL endpoints for RDF triple stores, as illustrated in Figure 1c), thus lowering the entry barrier to Linked Data for developers without prior experience with Semantic Web technologies and allowing to query existing RDF triple stores using GraphQL without manual setup steps.

Our work comprises the following contributions: an overview of related works exploring GraphQL as a query language for RDF (Section 2), an approach enabling the fully automatic bootstrapping of GraphQL endpoints for RDF triple stores (Section 3), an open source implementation thereof, called UGQL[1] (Section 4), based on HyperGraphQL [16], and a performance evaluation of this implementation (Section 5). We conclude with a summary of our findings and an outlook in Section 6.

[1] https://git.rwth-aachen.de/i5/ultragraphql

2 Related Work

The popularity of GraphQL in the Web engineering community has recently prompted several investigations into the integration of Semantic Web principles and general compatibility with Linked Data in the form of RDF data. Hereby, a principal challenge is the schemaless nature of RDF and the ability of users to dynamically create their own schema encodings, which makes it difficult to create a universal approach to make Linked Data queryable through GraphQL, which by default provides a static data schema (*GQLS*). We give an overview of approaches in the following, extending prior work by Taelman et al. [17].

Farré et al. introduced the GraphQL Metamodel (*GQLM*) [12] and corresponding RDF vocabulary, which allows for dataset enrichment with annotations, mapping RDF and RDFS concepts to GraphQL primitives. Based on these manual annotations, a GraphQL endpoint and corresponding data fetchers can be automatically generated. However, it requires both access to the underlying triple store and a modification of its data, as well as a deep understanding of the employed RDF ontologies and the GraphQL data model. This limits the general applicability of the approach, especially for usage with third-party triple stores.

Morph-GraphQL (*MGQL*) [18] is an approach that does not modify the data and generates the data fetchers from provided OBDA mappings (R2RML/RML), allowing the automatic generation of GraphQL endpoints for tabular datasets. The mapping is limited to the R2RML mapping vocabulary and is used to translate GraphQL queries to SQL to avoid the materialization of the data in RDF. RDF data is therefore not queryable with MGQL.

In contrast, GraphQL-LD (*GQLD*) [13] employs an ad-hoc query translation approach. It transforms a given GraphQL query to a corresponding SPARQL query by employing a user-provided semantic JSON-LD context [19], which maps defined keywords in the query to resource identifiers in the dataset. Since the semantic context is provided with each query, the adapter may be used with arbitrary triple stores. However, introspection becomes impossible since the adapter is oblivious to the data's schema. Subsequently, users are required to have detailed knowledge of the data schema of the underlying triple store, as well as a decent understanding of Semantic Web principles in general to define the required context object. GQLD query responses are plain JSON objects but may be interpreted as JSON-LD using the user-provided context object.

The commercial graph store *Stardog* [14] implements a hybrid approach, in which a data schema is optional. A GraphQL schema may be defined either manually, or be automatically bootstrapped from a provided RDFS and OWL ontology. It is then used by the Stardog endpoint to provide a GraphQL schema introspection endpoint, validate queries with the given typing information, optionally define custom translations of RDF values to GraphQL values and limit the user-accessible parts of the graph to the exposed schema. The automatic schema transformation is not configurable and limited to a predefined RDFS/OWL subset. If no schema is provided, introspection is not possible and no data validation or access control is enforced. The GraphQL query evaluation is based on the assumption, that the object at the query root refers to a type, while all other names refer to predicates. Additionally, a number of custom GraphQL directives are introduced in order to support several more advanced features such as

Table 1: Overview of GraphQL to RDF tools – '(✓)' denotes partial support

Features	**GQLM [12]**	**MGQL [18]**	**GQLD [13]**	**Stardog [14]**	**TopBraid [15]**	**HGQL [16]**	**UGQL**
Automatic Schema Extraction	(✓)	(✓)	-	(✓)	(✓)	-	✓
Schema Introspection	✓	✓	-	(✓)	✓	✓	✓
RDF-interpretable Results	-	-	✓	-	-	✓	✓
Filtering and Ordering	-	-	✓	✓	✓	(✓)	✓
Federated Query Support	-	-	✓	-	-	✓	✓
Mutation Support	-	-	-	-	✓	-	✓
License	None	Apache	MIT	Commercial	Commercial	Apache	Apache

filters and bindings. Notably, the handling of namespaced RDF predicates may require an additional understanding of SPARQL concepts or the structure of the underlying triple store. Results are provided as plain JSON and are not RDF-interpretable.

TopBraid [15] is another commercial product that enables querying RDF data using GraphQL. It tightly integrates schema definition and validation with SHACL shapes [20]. A GraphQL schema may either be automatically generated from an existing set of SHACL shapes or vice versa. TopBraid supports the semi-automatic creation of SHACL shapes from RDFS/OWL ontologies [21], which may in turn be used to generate a GraphQL schema from them. Datasets that use a different encoding or only a subset of the mapped vocabulary will lead to an incomplete schema and therefore a query may not be able to retrieve all available data [22]. Because of the required SHACL enrichment, the dataset must be fully accessible and modifiable by the developer; it thus can not be used for third-party triple stores [23]. The created SHACL shapes are used to validate any interaction with the triple store. Besides schema generation and schema introspection, TopBraid notably also supports GraphQL mutations, i.e. modifications to the underlying triple store through automatically generated CRUD operations, including corresponding types for the result and input of those functions. All approaches introduced up until here also support the filtering and ordering of the results. Lastly, query results are returned in plain JSON format and thus are not RDF-interpretable.

Finally, the Java-based open source project HyperGraphQL (*HGQL*) [16] is manually configured using a directive-annotated GraphQL schema, similar to both StarDog and TopBraid. It associates each type and field in the schema with a corresponding URI and supports the federated querying of multiple RDF triple stores. Only the types and fields defined in this *HyperGraphQL Schema* (HGQLS) are retrievable through GraphQL [24]. Based on the extended HGQLS, HGQL creates a GraphQL schema, query fields for all object types of the schema, and data fetchers for all schema entities. To assign each schema entity a responsible service, every type and field in the schema is extended with a directive linking to the responsible service. HGQL then supports schema introspection, as well as basic filtering and ordering operations and returns results using the JSON-LD format (i.e., including a semantic context object), so that the data can be interpreted as fully compliant RDF data.

While all described approaches provide viable tools to support querying RDF data sources using GraphQL, they simultaneously require a decent command of Semantic

Web principles and technologies, manual setup or knowledge of the structure of the data to be queried. In general, this limits their applicability to unknown data, ad-hoc data exploration and third-party data sources. The fully automatic bootstrapping of GraphQL endpoints for RDF triple stores remains an open challenge. Table 1 summarizes the relevant features of the described approaches (differentiating between no, partial and full support respectively). The last column introduces the set of features implemented by our work UltraGraphQL (*UGQL*), which we will present in the following.

3 Automatic GraphQL Bootstrapping

In order to simplify the usage and deployment of Linked Data GraphQL endpoints, we automate the setup and configuration through the introduction of a bootstrapping phase, consisting of an initial schema extraction and summarization based on a given dataset and a subsequent mapping of the extracted schema to a GraphQL schema configuration.

We first detail the two steps of the bootstrapping procedure illustrated in Figure 2, before describing our implementation UltraGraphQL and its usage in more detail.

3.1 Schema Extraction and Summarization

RDF is by design a schemaless data model but allows for the flexible definition of custom schema semantics using a variety of ontologies. Notable examples include RDFS and OWL [21], which are commonly used to formalize class semantics and relations thereof, but a large variety of schema encoding approaches exists in practice. Therefore, we strive to support a commonly used subset of RDFS and OWL by default, while allowing the user to adapt the schema extraction if additional terms are necessary.

An early approach to schema extraction by Matono et al. [25] employs a relatively minimal set of concepts to approximate the full data schema, specifically *rdfs:Class*, *rdfs:Property*, *rdfs:subClassOf* and *rdfs:subPropertyOf*, in the context of search indexing. Florenzano et al. [26], Lohmann et al. [27,28], Dudáš et al. [29] and Benedetti et al. [30,31] follow similar approaches to determine classes and properties, either through their instantiation in the dataset, or their explicit concept annotation, however, they mainly focus on schema extraction for data structure visualization. Kellou-Menouer et al. [32] propose an approximate schema discovery approach based on hierarchical clustering instead of data annotations.

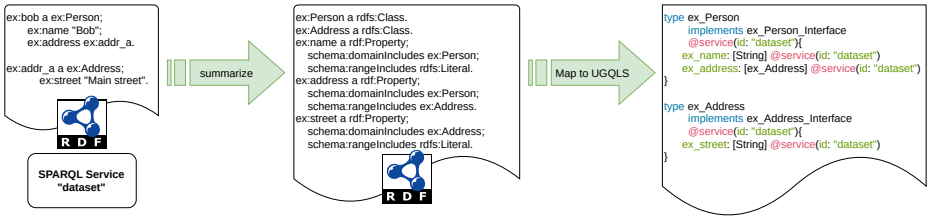


Fig. 2: Two phases of GraphQL bootstrapping: Schema summarization & mapping

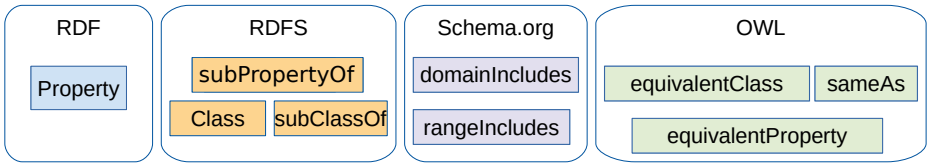


Fig. 3: RDF vocabulary employed in the extracted data schema

As outlined in Section 2, TopBraid [33] supports the semi-automated creation of SHACL data shapes from OWL and RDFS ontologies, which in turn can then be interpreted as data schema. Stardog [14] directly interprets OWL and RDFS ontologies as data schemas, regardless of their actual use in the data. Ontology2GraphQL [34] follows a similar approach to create an HGQLS configuration from ontologies, following the ICDD (Information Container for Data Drop) standard [35].

To ensure that the automatically extracted schema includes only types and fields instantiated in the respective RDF data sources, we recently proposed a schema extraction and summarization approach [36], which allows for the flexible extraction of instantiated schema from a given RDF data set through configurable SPARQL 1.1 path queries. We adopt this approach to ensure that only the instantiated data schema is extracted, leaving out ontological concepts that are defined but never used. The extracted schema is then expressed using a fixed RDF vocabulary as depicted in Figure 3, such that the subsequent GraphQL schema mapping can be limited to those concepts. Other semantic concepts that are used in the data but not included in the extraction vocabulary are eventually accessible through extracted schema. For example, rdf:List is a rdfs:Class and is therefore queryable as an object with rdf:first and rdf:rest as fields. The approach further summarizes defined schema concept equivalences, enabling cross dataset and cross ontology query resolution. To provide support for multiple RDF backend stores, the extraction query is simultaneously enriched with service-specific information and each service is queried separately. This extracted data schema can then be used for the automatic generation of a corresponding GQLS, using a suitable schema mapping. The quality of the extracted data heavily depends on the chosen extraction query and mapping and may require adjustment to the underlying RDF data.

3.2 Schema Mapping

In order for a GraphQL endpoint to provide schema introspection capabilities, a predefined GQLS is required. Therefore all primitives of the previously fixed vocabulary depicted in Figure 3 are aligned with corresponding ones in GQLS, as summarized in Table 2. Fundamentally, classes in RDF are mapped to object types in GraphQL, while RDF properties become GraphQL fields of those object types which may occur as part of their RDF domain. The output types of these fields are analogously defined by the property's range, i.e., *String* for literal values or the object types corresponding to the defined classes. [2] If multiple ranges are defined for a property in the data schema, a

[2] https://git.rwth-aachen.de/i5/ultragraphql/-/blob/master/docs/schema_mapping.md

Table 2: Overview of the mapping between RDF and UGQL

RDF	UGQL
Class	Object Type + Interface Type
Property	Field
Domain	Domain(Object) of Field
Range	Output Type of Field
Literal	String
SubClassOf	Interface Type + implements
SubPropertyOf	Add Domain and Output Type of Sub-Property to Super-Property
EquivalentClass	Mutual implements + directive
EquivalentProperty	Merging Domain and Output Type of Both Fields + directive

corresponding GraphQL interface type is generated, combining all acceptable atomic types. Inspired by the usage of prefixes in SPARQL, the names of generated schema entities may use namespace abbreviations to increase their readability.

To express the extracted subclass and equivalence relationships between schema entities, we employ GraphQL interfaces. For each object type, a corresponding interface type is defined, which is then implemented by the object type itself (in compliance with RDFS's semantics that all classes are a subclass of themselves) and all of its subclasses. The equivalence relation between classes is then expressed as a mutual subclass relationship (following OWL semantics) and mapped to a mutual interface implementation. Since GQLS provides no primitive to express relations between fields, we materialize subproperty and equivalence relations. Hence, if a given field occurs on an object type, all of its super-properties are added as well and each field's output types are extended to include the range or its super-properties. Property equivalence is analogously modeled as a mutual sub-property relationship. Accounting for frequent misuse [37], we further treat OWL *sameAs* relations in the extracted schema analogously to equivalence definitions. As seen by the equivalence relations, a perfect mapping to GQL is not possible since GQL has a limited set of supported features, resulting in schema features that are natively not supported by GraphQL. To circumvent this limitation, the mapping materializes the unsupported features to allow query writing without any further knowledge and shifting the underlying logic to the query translation.

3.3 Automatic GraphQL to SPARQL translation

After generating a GQLS configuration from the extracted data schema using the above mapping, we can now deploy an adapter instance to handle incoming GraphQL queries during the translation phase. Figure 4 illustrates a concrete example of a query requesting information about a person and the corresponding JSON-LD response returned by the UGQL adapter. Depending on the underlying UGQLS, the query is translated to SPARQL queries covering the distribution of data across triple stores as defined in the UGQLS. [3] The number of generated queries depends on the structure of the query in correlation

[3] https://git.rwth-aachen.de/i5/ultragraphql/-/blob/master/docs/translation_phase.md

with the distribution of triple stores on the queried entities. The results are then merged, creating a unified result across all supported triple stores and enriched with contextual information about the IRIs of the queried entities. Given concepts for schema extraction, mapping and translation, we detail the realization of UGQL and its usage in the following.

4 UltraGraphQL Bootstrapping

To simplify the deployment of GraphQL endpoints for RDF triple stores, we present *UltraGraphQL* (UGQL). Extending the features and codebase of HyperGraphQL [16], UGQL is an open source adapter supporting a fully automatic bootstrapping process based on a flexible and configurable SPARQL schema extraction, mapping and query translation approach. UGQL further supports elaborate data filtering and ordering, as well as GraphQL mutations, by automatically generating corresponding mutator functions for the GQLS types and fields. As such, to the best of our knowledge, it is not only the first available tool supporting fully automatic bootstrapping but also the first such open source tool to support mutations. UGQL thus enables developers without prior knowledge of data schema or Semantic Web technologies in general to read and write Linked Data using GraphQL.

The software architecture of UGQL, illustrated in Figure 5, is designed to mirror the two distinct phases *bootstrapping* and *translation* described in the previous section. Green boxes indicate existing work reused from HGQL, orange boxes show adapted prior work and red boxes point to entirely novel components, as described in the following.

During the initial *bootstrapping* phase, an endpoint configuration provided in step 1 determines the automatic data schema extraction from the underlying RDF triple stores in step 2. The extracted data schema is then mapped to an UltraGraphQL schema (UGQLS) in step 3, which is cached and may either be manually adapted if needed or automatically regenerated at any time. The UGQL schema extends HGQLS by supporting unions, interfaces, and multiple services per schema entity. Therefore, any valid HGQLS is a valid UGQLS. Both schema extraction query and mapping phases are configurable, allowing to modify and extend the schema to be extracted from the data as well as the mapping of RDF vocabulary terms to GraphQL concepts. Detailed documentation of the UGQLS syntax and options can be found in the project repository[1].

For the *translation* phase, the generated UGQL schema is passed to the UGQL instance to initialize the GraphQL endpoint in step 4. At this point, the UGQL adapter is

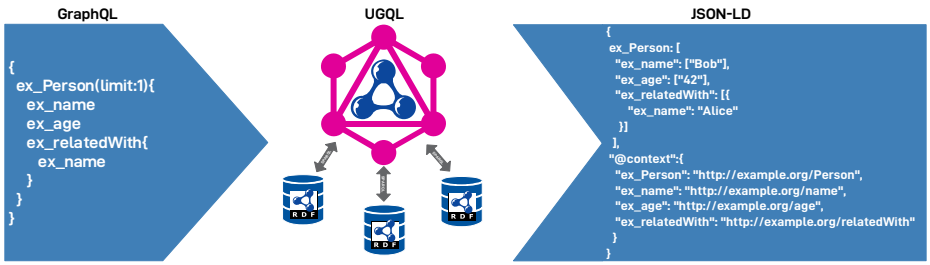


Fig. 4: Querying multiple RDF stores through UGQL

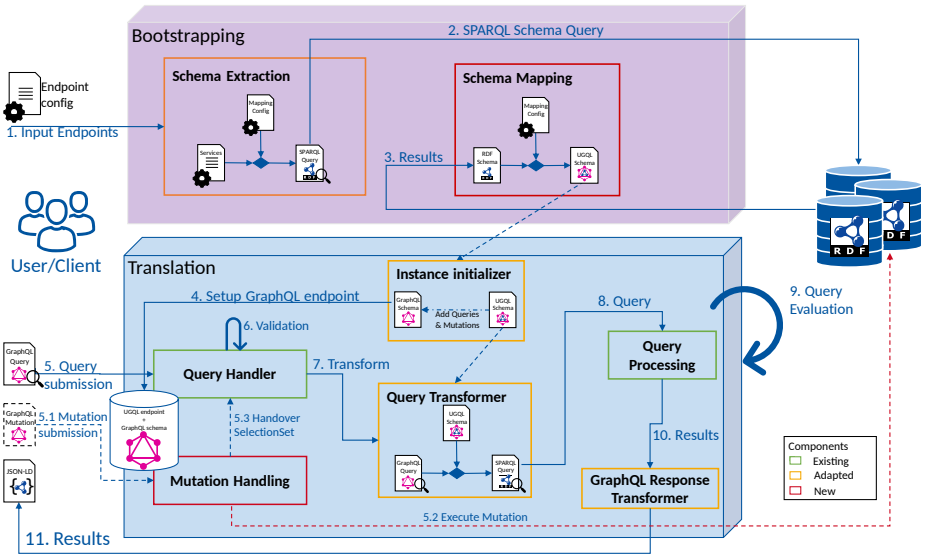


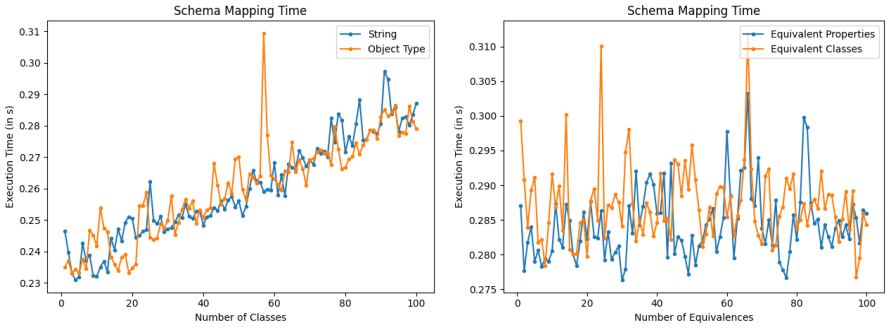
Fig. 5: Control flow through UltraGraphQL's software components

able to accept GraphQL requests. Upon query submission (step 5), it is first validated for compliance with the GraphQL schema (step 6) before being transformed into a corresponding SPARQL query using additional HGQLS information (step 7). Subsequently, it is executed against the respective underlying SPARQL endpoints (steps 8–10). Analogously, mutation requests (5.1) are validated by a mutation handler before being executed directly against the underlying triple stores (step 5.2 via SPARQL Update). The mutation action is only executed at one triple store; a mutation SelectionSet is executed against all triple stores. If successful, the resulting query (after mutation) is then passed to the regular GraphQL query handler for further processing as before (step 5.3). Finally, the SPARQL response is transformed into a valid GraphQL response in JSON-LD format and returned to the user (step 11). Further details on UGQL's advanced features such as support for multiple endpoints, mutations and filtering and ordering can be found in the project documentationItem ?.

5 Performance Evaluation

To measure the performance of UGQL, we conducted manual and quantitative evaluations of the proposed schema mapping process and its applicability, as well as comparative qualitative and quantitative evaluations of UGQL properties such as query size, execution time, and response size during translation, versus the baseline of HGQL (*v1.0.3*) and plain SPARQL queries. Instructions to reproduce our results, all data, queries and code employed, may be found in our project repository[4]. All practical experiments were conducted on a machine running *Ubuntu 18.04* with *16GB* RAM on an *Intel Core*

[4] https://git.rwth-aachen.de/i5/ultragraphql/-/tree/master/evaluation



(a) 5 properties per class with range String or a random Object Type

(b) 100 classes with 5 properties each and n randomly defined Property or Class equivalences in total

Fig. 6: Quantitative evaluation of the schema mapping execution time

i5-5300U with hyper-threading enabled. After an initial warm-up period, each test was repeated 100 times and the results averaged. In the following, we structure our evaluation according to the concept section.

5.1 Bootstrapping Evaluation

Since the schema extraction was only slightly adjusted to include *rdfs:Literal* range information in the schema, we refer to a corresponding evaluation in our prior work [36] and focus on the remaining mapping process in this evaluation.

To ensure the qualitative correctness of the mapping process as described in Section 3.2, the GQLS generation was first verified using corresponding unit tests, as documented in the project repository[5], and then manually validated for correctness with different data schemas. We further quantitatively evaluated the runtime of the generation process in two experiments, controlling the impact of schema size, i.e., the number of classes and properties, and the number of relations between classes: (a) the number of classes and properties was increased with each step with String as the range for one test case and a random class for the other test case, (b) a schema with 100 classes and five properties per class was used and in each step one equivalence relation depending on the test case was added to evaluate the influence of the equivalence relations. The results, depicted in Figure 6, indicate that the time to generate the mapping depends primarily on the size of the schema, and less on specific relations between these entities. Even though rapid schema mapping is only of limited importance for the usability of UGQL since it only affects the bootstrapping process during deployment, the measured generation runtime for a schema with 100 classes and a total amount of 500 fields remain below the responsiveness threshold [38] of $300ms$ and are likely orders of magnitude faster than a manual GQLS mapping process. As such, our evaluations confirm that UGQL fulfills our initially defined bootstrapping requirements and allows for the automated generation

[5] https://git.rwth-aachen.de/i5/ultragraphql/-/tree/master/docs/evaluation

of GraphQL schema configurations, eliminating the need for manual user interaction. In the next section, we evaluate the performance of UGQL in the translation phase.

5.2 Translation Evaluation

To evaluate UGQL's performance in the translation phase, we compare its runtime, query and result sizes to HGQL (*v1.0.3*) and plain SPARQL queries on various tasks. Similar to our evaluation of the bootstrapping phase, we first conduct a qualitative evaluation based on a limited number of fixed queries, and second a quantitative evaluation, subsequently querying for a) an increasing number of properties on a given entity and b) increasing query depth. We employ an *Apache Jena Fuseki*[6] (*v3.7.0*) in-memory RDF store to serve as SPARQL endpoint throughout the evaluation.

For the qualitative evaluation, we execute 4 manually crafted queries[7] against a dataset[8] with 9974 triples, 1300 distinct subjects, 8 unique properties and 2 unique classes. Starting with a UGQL query as a reference, a corresponding HGQL query was derived by adapting the entity naming to its requirements, and the SPARQL query was directly generated by HGQL. UGQL and HGQL were both configured with the same schema that was extracted using UGQL's bootstrapping process, merely manually limited to features supported by both softwares. UGQL was then deployed in two configurations, the first with the dataset loaded into an internal in-memory store and the second with the external Fuseki server, to measure the effect of the additional HTTP communication. HGQL used the same external Fuseki server. SPARQL was evaluated both with the common JSON response format and a more compact CSV representation. The results in Table 3 show that the size of UGQL queries is consistently significantly smaller than that of the respective SPARQL query. Similarly, the size of UGQL's and HGQL's results shows a 59–81% reduction compared to the SPARQL JSON baseline, and between 41% reduction and 112% increase compared to SPARQL's CSV response format, which is however harder to parse on the client-side and therefore less applicable to the intended use-case. The size reduction of GraphQL responses results from both a more compact JSON structure compared to SPARQL's response format and the merging of redundant information into lists. The reductions in size however come at the cost of a significantly increased response time by up to an order of magnitude, comparing UGQL Fuseki with the SPARQL JSON baseline in case of query 1. For reasonably sized results (query 1–3), the response time however remained well below the human responsiveness threshold [38] of $300ms$ and therefore acceptable for our intended use case scenario. The increase in execution time from HGQL to UGQL can be explained by extended type checking, introduced by added features like interfaces and equivalence relations requiring additional type checks during the query translation for any query entity. Furthermore, the GraphQL library employed in the HGQL and UGQL code base was identified as a main cause of overhead by introducing various internal transformations of the results during query response generation. It is thus clear to us that a direct result build-up, i.e., without relying on the GraphQL library used internally, would perform significantly faster and is left for future work.

[6] https://jena.apache.org/documentation/fuseki2/
[7] https://git.rwth-aachen.de/i5/ultragraphql/-/tree/master/evaluation/queries/one_service
[8] https://git.rwth-aachen.de/i5/ultragraphql/-/blob/master/evaluation/data/raw/persons_and_cars.ttl

Table 3: Qualitative comparison of UGQL, HGQL and SPARQL

Metric	Query	UGQL		HGQL	SPARQL	
		Standalone	Fuseki	Fuseki	CSV	JSON
Query Size	1	**64 B**		68 B	168 B	
	2	**78 B**		82 B	280 B	
	3	**187 B**		202 B	260 B	
	4	**155 B**		160 B	258 B	
Result Size	1	30.1 KB			**17.4 KB**	73.3 KB
	2	10.4 KB			**4.9 KB**	33.4 KB
	3	**5.7 KB**			8.1 KB	22.6 KB
	4	**2.5 MB**			4.7 MB	13 MB
Latency / Response Time	1	139 ms	180 ms	88.4 ms	**10 ms**	19 ms
	2	68.1 ms	103 ms	50.9 ms	**8 ms**	14.5 ms
	3	48.3 ms	64.3 ms	42.7 ms	**7.3 ms**	13.2 ms
	4	3,530 ms	4,640 ms	2,970 ms	**240 ms**	715 ms

For the quantitative evaluation, we measure the impact of query growth on the response time and query result size, both in terms of the number of queried fields and the tree depth of the query. To analyze the effect of the tree depth of the query, i.e., the influence of nested queries, we iteratively request all persons and all other persons they are related with over up to n (up to 50) hops from a dataset[9] with 50 persons in total, each of which is the subject of one *ex:relatedWith* relation to another random one of them. To analyze the effect of the number of queried fields, we query all persons together with the first n (up to 50) associated fields from a dataset[10] of 1000 persons with 1000 fields with a random literal value of length ten each.

The results, depicted in Figure 7, indicate that UGQL appears to add an approximately constant multiplicative factor of execution, i.e., response time overhead in both scenarios. UGQL further consistently returns an approximately constant multiplicative factor smaller result sizes then SPARQL with JSON result format, but similarly approximately constant multiplicative factor larger result sizes then SPARQL with CSV result format. Both execution time and response size further increase approximately linearly with the number of queried fields for all approaches and execution time appears to correlate strongly with overall result size in all cases. Nevertheless, with an overhead factor of approximately 3, query depth has a significantly larger impact on UGQL's execution time overhead over plain SPARQL then the number of fields queried with a factor of approximately 2. Manual investigation revealed a majority of the overhead to be caused by the final result transformation in the GraphQL library used internally by HGQL and UGQL, taking up $63,4\%$ of the whole execution time in case of the last query of the growing depth test. This finding further supports the notion that a direct result transformation could significantly reduce overhead and therefore response times in future work. Overall, UGQL is particularly well-suited for usage in applications with

[9] https://git.rwth-aachen.de/i5/ultragraphql/-/blob/master/evaluation/data/nested_person_data.ttl
[10] https://git.rwth-aachen.de/i5/ultragraphql/-/blob/master/evaluation/data/growing_field_data.ttl

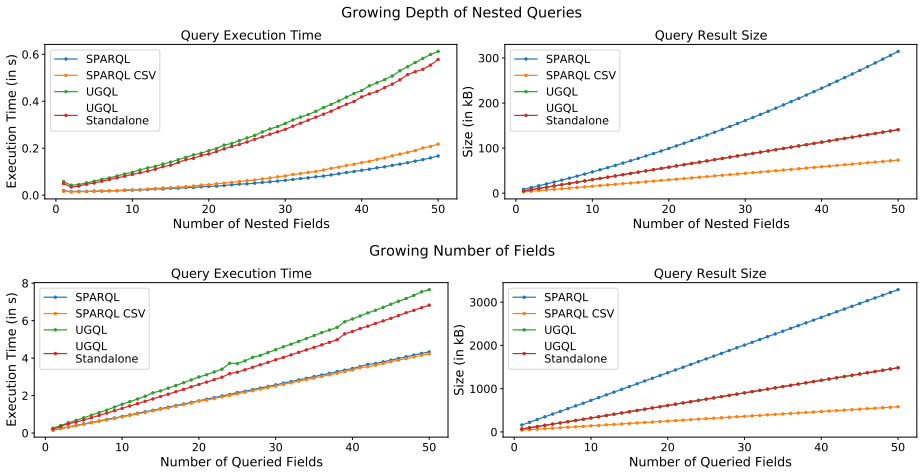


Fig. 7: Impact of query growth on response time and result size

typically small result sizes, as it is able to resolve such queries with a latency under the human responsiveness threshold. In conjunction with automatic schema generation and bootstrapping and the implemented extensions over HGQL, UGQL provides users who are unfamiliar with Semantic Web technologies a simple and responsive query endpoint to access RDF data with the benefit of structured and reduced query results. It is therefore also particularly suited for applications in mobile computing or Web development scenarios.

6 Discussion and Conclusion

In this paper, we presented a conceptual design and tool support to automatically bootstrap GraphQL endpoints for existing RDF triple stores. The main argument for such an adapter is that RDF and SPARQL are verbose and unfamiliar to most Web developers, while GraphQL query language is becoming increasingly popular with them. This is due to its suitability for resource-constrained mobile devices and developer-friendliness by introspection capabilities often bundled with service API endpoints. It is especially advantageous if developers are not entirely aware of the exact structure of the data they are querying. In contrast, querying RDF data requires detailed knowledge of the underlying schema in order to write queries at all. Another criticism regarding SPARQL results is that they contain too much metadata and redundant information, resulting in an increased data exchange and additional computational costs. In contrast, GraphQL allows the explicit specification of the required data fields and employs a more condensed tree-shaped result format. As such, it is better suited for mobile applications, since it results in reduced data exchange and computational cost. Even though existing approaches introduced in Section 2 allow to query RDF data with GraphQL, they all need a predefined schema, schema-enhanced dataset or prior knowledge of the data structure to function properly. Due to the schemaless nature of RDF, frequent changes can occur,

making manual schema generation tiresome and nearly impossible to manually provide for large and dynamic datasets.

To this end, we introduced UltraGraphQL, a fork of HypherGraphQL extended with the capability to automatically generate a GraphQL schema for existing RDF stores, as well as additional data filtering, ordering and mutation capabilities. The on-demand schema summarization of RDF datasets, using adaptable SPARQL queries, allows extracting the instantiated schema of the dataset, enabling for querying RDF data without prior knowledge of the underlying data structure. We proposed a flexible mapping approach enabling the automatic generation of a UGQLS configuration based on the extracted schema. Furthermore, the feature set of HGQL was extended to support ontological equivalence relations and support for multiple services per schema entity. Summarizing the schema of multiple services into one UGQL schema allows to query the endpoint without knowing the location of the data. In addition to the automatic bootstrapping, HGQL was extended with GraphQL mutation capabilities, providing an introspection-supported interface for data alteration. Notably, this enables developers without knowledge of Semantic Web technologies to independently and automatically bootstrap GraphQL endpoints to read and write Linked Data, even for frequently changing triple stores.

The evaluation has shown that in most cases, UGQL results in a 59–81% response size reduction compared to the SPARQL JSON baseline, while only introducing a tolerable single-digit factor of response time overhead. Notably, the latency remained well below the human responsiveness threshold of $300ms$ for our representative sample queries. The reduced query and result sizes increase the efficiency of RDF data in mobile applications by reducing the amount of transmitted data and computational cost of data conversion. We therefore encourage providers of RDF triple stores to use our tool to significantly increase the usability of their offerings for mobile and web developers.

Some future work remains to be solved. First of all, we are currently preparing a user study as quantitative assessment for our claim to improve usability for developers. On the one hand, we plan to ask developers familiar with SPARQL to recreate some queries with GraphQL and rate the gains or overheads. On the other hand, we want to evaluate with web developers on how natural the generated interfaces feel. Regarding performance, our measurements indicates that the response time overhead of UGQL may be significantly reduced by eliminating redundant transformations in the internally used GraphQL library, potentially cutting response times in half. Secondly, mutation support could be enhanced to support live schema updates by analyzing the input of mutations, enabling the modification of types and fields at runtime. Lastly and most importantly, we are currently preparing a user study complementing our conducted performance evaluations, to validate the claimed promise of making RDF triple stores more accessible for Web and mobile application developers via GraphQL.

Overall, we are convinced that the presented open source tool is a valuable addition to the Semantic Web toolset. As GraphQL continues to gain popularity among developers, UltraGraphQL is ideally positioned to unfold the true potential of semantically enriched data in various application areas.

Acknowledgments. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

References

1. J. Pennekamp, R. Glebke, M. Henze, T. Meisen, C. Quix, R. Hai, *et al.*, "Towards an Infrastructure Enabling the Internet of Production," in *Proceedings - 2019 IEEE International Conference on Industrial Cyber Physical Systems, ICPS 2019*, 2019.
2. L. Gleim, J. Pennekamp, M. Liebenberg, M. Buchsbaum, P. Niemietz, S. Knape, *et al.*, "FactDAG: Formalizing Data Interoperability in an Internet of Production," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3243–3253, 2020.
3. D. Wood, M. Lanthaler, and R. Cyganiak, "RDF 1.1 Concepts and Abstract Syntax." W3C, 2014.
4. A. Seaborne and S. Harris, "SPARQL 1.1 Query Language." W3C, 2013.
5. D. Booth, "Toward Easier RDF." W3C Workshop on Web Standardization for Graph Data, 2019.
6. P. Lisena, A. Meroño-Peñuela, T. Kuhn, and R. Troncy, "Easy Web API Development with SPARQL Transformer," in *The Semantic Web – ISWC 2019*, pp. 454–470, Springer International Publishing, 2019.
7. P. Lisena and R. Troncy, "Transforming the JSON Output of SPARQL Queries for Linked Data Clients," in *Companion Proceedings of the The Web Conference 2018*, WWW '18, pp. 775–780, 2018.
8. W. Van Woensel, S. Casteleyn, E. Paret, and O. De Troyer, "Transparent Mobile Querying of Online RDF Sources Using Semantic Indexing and Caching," in *Web Information System Engineering – WISE 2011*, pp. 185–198, Springer Berlin Heidelberg, 2011.
9. Facebook Inc., "GraphQL. Working Draft," 2020.
10. G. Brito, T. Mombach, and M. T. Valente, "Migrating to GraphQL: A Practical Assessment," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 140–150, 2019.
11. O. Hartig and J. Pérez, "Semantics and complexity of GraphQL," in *Proceedings of the 2018 World Wide Web Conference*, pp. 1155–1164, 2018.
12. C. Farré, J. Varga, and R. Almar, "GraphQL Schema Generation for Data-Intensive Web APIs," in *Model and Data Engineering*, pp. 184–194, Springer International Publishing, 2019.
13. R. Taelman, M. Vander Sande, and R. Verborgh, "GraphQL-LD: Linked Data Querying with GraphQL," in *Proceedings of the 17th International Semantic Web Conference*, pp. 1–4, 2018.
14. Stardog Union, "Stardog 7: The Manual." https://www.stardog.com/docs/#_graphql_queries, 2020.
15. TopQuadrant, "Updating RDF Graphs with GraphQL." https://www.topquadrant.com/technology/graphql/graphql-mutations/.
16. Semantic Integration Ltd., "HyperGraphQL Git Respository." https://github.com/hypergraphql/hypergraphql, 2018.
17. R. Taelman, M. Vander Sande, and R. Verborgh, "Bridges between GraphQL and RDF," in *W3C Workshop on Web Standardization for Graph Data*, 2019.
18. 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*, vol. 30, no. 06, pp. 785–803, 2020.
19. M. Lanthaler, M. Sporny, and G. Kellogg, "JSON-LD 1.0." W3C rec., 2014.
20. D. Kontokostas and H. Knublauch, "Shapes Constraint Language (SHACL)." W3C rec., 2017.

21. W3C OWL Working Group, "OWL 2 Web Ontology Language." https://www.w3.org/TR/owl2-overview/, 2012.
22. I. Polikoff, "From OWL to SHACL in an automated way." https://www.topquadrant.com/from-owl-to-shacl-in-an-automated-way/, 2018.
23. H. Knublauch, D. Allemang, and S. Steyskal, "SHACL Advanced Features." https://www.w3.org/TR/shacl-af/.
24. Semantic Integration Ltd., "HyperGraphQL." https://www.hypergraphql.org/, 2018.
25. A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura, "An Indexing Scheme for RDF and RDF Schema based on Suffix Arrays," in *Proceedings of SWDB'03*, pp. 151–168, 2003.
26. F. Florenzano, D. Parra, J. L. Reutter, and F. Venegas, "A Visual Aide for Understanding Endpoint Data," in *Proceedings of the Second International Workshop on Visualization and Interaction for Ontologies and Linked Data co-located with the 15th International Semantic Web Conference, VOILA@ISWC*, vol. 1704, pp. 102–113, 2016.
27. S. Lohmann, V. Link, E. Marbach, and S. Negru, "Extraction and Visualization of TBox Information from SPARQL Endpoints," in *Proceedings of the 20th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2016)*, vol. 10024 of *LNAI*, pp. 713–728, Springer, 2016.
28. M. Weise, S. Lohmann, and F. Haag, "LD-VOWL: Extracting and Visualizing Schema Information for Linked Data," *Visualization and Interaction for Ontologies and Linked Data (VOILA! 2016)*, p. 120, 2016.
29. M. Dudáš, V. Svátek, and J. Mynarz, "Dataset Summary Visualization with LODSight," in *International Semantic Web Conference*, pp. 36–40, Springer, 2015.
30. F. Benedetti, S. Bergamaschi, and L. Po, "Online Index Extraction from Linked Open Data Sources," in *Proceedings of the Second International Workshop on Linked Data for Information Extraction (LD4IE 2014) co-located with the 13th International Semantic Web Conference (ISWC 2014)*, pp. 9–20, 2014.
31. F. Benedetti, S. Bergamaschi, and L. Po, "Visual Querying LOD sources with LODeX," in *Proceedings of the 8th International Conference on Knowledge Capture*, p. 12, ACM, 2015.
32. K. Kellou-Menouer and Z. Kedad, "Schema Discovery in RDF Data Sources," in *International Conference on Conceptual Modeling*, pp. 481–495, Springer, 2015.
33. TopQuadrant, "Querying RDF Graphs with GraphQL." https://www.topquadrant.com/graphql/graphql-queries.html, 2017.
34. J. Werbrouck, "Ontology2GraphQL." https://github.com/JWerbrouck/Ontology2GraphQL, 2019.
35. J. Werbrouck, M. Senthilvel, J. Beetz, and P. Pauwels, "Querying Heterogeneous Linked Building Data with Context-expanded GraphQL Queries," in *Proceedings of the 7th Linked Data in Architecture and Construction Workshop*, vol. 2389, pp. 21–34, 2019.
36. L. C. Gleim, M. R. Karim, L. Zimmermann, O. Kohlbacher, H. Stenzhorn, S. Decker, and O. Beyan, "Enabling ad-hoc reuse of private data repositories through schema extraction," *Journal of Biomedical Semantics*, vol. 11, no. 1, p. 6, 2020.
37. P.-H. Paris, "Assessing the Quality of owl:sameAs Links," in *The Semantic Web: ESWC 2018 Satellite Events*, pp. 304–313, Springer International Publishing, 2018.
38. J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.