Querying Heterogeneous Linked Building Data with Context-expanded GraphQL Queries*

Jeroen Maurits Werbrouck[1,2], Madhumitha Senthilvel[1], Jakob Beetz[1][0000−0002−9975−9206], and Pieter Pauwels[2][0000−0001−8020−4609]

[1] Faculty of Architecture, RWTH Aachen University, 52062 Aachen, Germany
[2] Dept. of Architecture and Urban Planning, Ghent University, 9000 Ghent, Belgium

Abstract. Linked Data in the construction industry is a topic gaining serious interest over the last years. However, this interest remains largely academic and has not sparked much adoption of web technologies in the field. To nourish adoption of Linked Data in practice, access to the data has to be made more easy. SPARQL, the recommended RDF query language, has proven very powerful for retrieving and updating RDF datasets. However, due to its verbosity and complexity, it is often considered a threshold for developers to implement in their tools. In this paper, we compare SPARQL with Linked Data querying languages that extend the GraphQL syntax, in context of querying building datasets: HyperGraphQL and GraphQL-LD. Since it went open source in 2015, GraphQL has been adopted by a large community of developers, partly due to its elegance and conciseness. As a use case, the queries are performed on an RDF-based multi-model that relies on the recent ISO standard ICDD (ISO 21597). ICDD interlinks information on a sub-document identifier level, based on OWL. It is aimed at the industry in that it adds a layer of Linked Data to documentation formats that are still widely used in practice: IFC, spreadsheets, imagery etc. Therefore, it is considered a tuned use case for comparing these different RDF query languages.

Keywords: Linked Data · ICDD · SPARQL · GraphQL.

1 Introduction

The advent of Building Information Modelling (BIM) has brought a revolution to the way a construction project is designed, constructed and maintained. BIM addresses the need for an 'information hub' that contains and manages information on architecture, engineering and 'Mechanical, Electrical, Plumbing' (MEP) topics. Through BIM, professionals in these disciplines are able to communicate in a much more streamlined way than before.

* Supported by the European Union's Horizon 2020 Research and Innovation Program under grant agreement No 820773, and the Ghent University Special Research Fund (BOF).

In an ideal building industry, linking of available information happens, and happens on a data level. This has been proposed multiple times since the late 2000s [2, 1] and is therefore actively researched by multiple researchers. One of the most known achievements is the translation of the IFC (Industry Foundation Classes) schema into ifcOWL; its Linked Data equivalent [2, 5]. A more recent trend is to focus on smaller, modular ontologies, often aligning with the Building Topology Ontology (BOT) [6, 7]. Each one of these smaller ontologies targets one aspect of the building industry, yielding a flexible and functional way for knowledge modelling when combined with one another.

In the building industry field, however, the use of RDF (Resource Description Framework), OWL (Web Ontology Language) and other web technologies is still largely an academic topic, and not all pitfalls are solved to convince BIM software providers and the industry to dive into the world of Linked Building Data. More achievable at the moment is a cross-document way of linking sources and BIM documents, instead of implementing the project entirely in RDF. However, this is not applied very frequently either. To improve an ontology-based, link set-oriented approach [13], the Information Container for Data Drop (ICDD)[3] is in its final stages of international standardisation as ISO 21597. ICDD provides a standard way to link documents with one another. It is a container-like structure that uses a `Container` OWL ontology and a `Linkset` OWL ontology that contain the definitions to describe and link document data with nodes in RDF graphs or other document data. Using the provided terminologies, information linking on sub-document level could be reached. This means that, for example, a single cell in a spreadsheet can be linked to an identifier in an IFC document and a pixel zone in an image. The resulting instance graphs describe metadata and links between both internal sources included in the ICDD container folder, or external data sets, located somewhere on the web.

In the same ideal world, RDF data is queried by use of the official W3C recommendation SPARQL (SPARQL Protocol and RDF Query Language)[4]. However, SPARQL is often considered too high a threshold for developers to implement in their tools as a querying language [14], which is why different initiatives are set up to lower this threshold, often based on JSON-LD[5] contexts. These initiatives focus either on direct implementation in programming languages, such as LDflex[6] for Javascript or on language-independent solutions such as GraphQL extensions for Linked Data [12]: HyperGraphQL[9] and GraphQL-LD [11]. GraphQL[3] was originally constructed by Facebook Inc. in 2012 and developed as an open standard from 2015 on. It can be both used for data querying and as an API querying language. A general overview on the main differences between HyperGraphQL and GraphQL-LD for application in cloud-based collaboration networks for the building industry is given in [16]. With respect to cloud-based applications, [16] conclude that GraphQL-LD is a more flexible approach, due to its server- and

[3] https://www.iso.org/standard/74389.html
[4] https://www.w3.org/TR/sparql11-query/
[5] https://www.w3.org/2018/jsonld-cg-reports/json-ld/
[6] https://github.com/RubenVerborgh/LDflex

schemaless deployment. In this paper, we investigate the potential of both HyperGraphQL and GraphQL when querying a fixed, local dataset. The discussed use case differs from the one in [16] in that it does not focus on cloud-based services, but on querying the available information in a link set container, namely ICDD, as a dump file of the available project information. Using ICDD, the ontologies that are used within a container are known, and most data can be queried through a single endpoint.

2 Related work

2.1 Information Container for Data Drop

In its own words, ICDD has been developed *'in response to the need of the construction industry to handle multiple documents as one information delivery or data drop.'* It is a successor of the Dutch COINS standard and fits in a tradition of ontology-based management of multi-models [8, 4, 13]. Information elements within a container can be linked with one another and with external data by making use of Semantic Web technologies. These relationships strengthen the semantic quality of the container, because they allow to refer to other data, even if this data is not structured following RDF principles. For the construction industry, typically slow in its adoption of new technologies, ICDD provides a bridge between documents structured in previously incompatible data formats. From an industry point of view, ICDD containers could be considered as a package used to send a linked file repository, to store a particular version or state of a complex project. Although it allows to refer to external documents, the standard focuses on the description of internal documents, assuming that this scenario, where all referred documents are contained in the ICDD folder, occurs more frequently. In such situation, an efficient and accessible way to query the available project data stored in the container could be of use. From a research point of view, ICDD might provide an interesting way to store and link data formats for which an RDF representation is not considered an efficient solution, such as imagery, point clouds or geometric representations. A distributed web-based ecosystem for building projects may use the ICDD ontologies to semantically link RDF building data with non-RDF, document-based project information, rather than focusing on creating a dump folder (although this might be an interesting side effect) [15].

A container in .icdd format (using a ZIP-compression) consists of a fixed folder structure, with separations between ontologies (*Ontology resources*), documents (*Payload documents*), specific linksets (*Payload triples*) and content description (*index.rdf*) (Fig. 1). Graphs that describe a container's content and links between documents are stored in different graphs, but refer to the same URIs of documents. A visual graph depicting the link between two documents is given in Fig. 2.

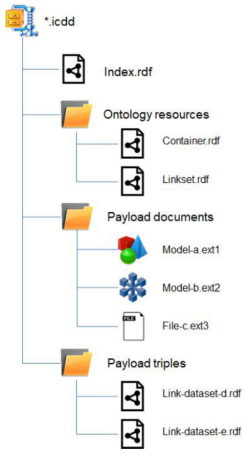

Fig. 1. Hierarchy of folders and files in a container (source: Information container for data drop - Exchange specification - Part 1: Container)

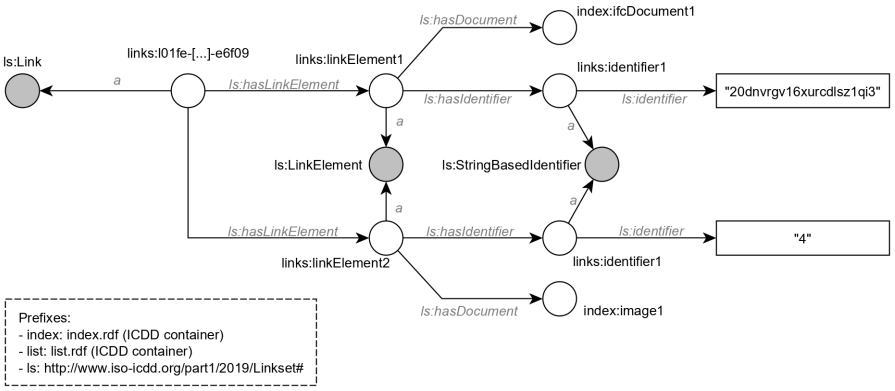

Fig. 2. RDF-based link between identifiers in two different documents: an IFC file and a spreadsheet

2.2 GraphQL in a Linked Data context

GraphQL[7] is both a data query language and a specification to minimise and optimise data transfer over the web over a GraphQL API. It is specifically aimed at improving client-server interactions and has been adopted by multiple large players on the web. Its core exists of a schema that describes the data to use: its class, its fields and the classes or datatypes those fields may refer to. From this perspective, these schemes can be visualised as graphs. However, they should not be interpreted as RDF graphs, as the types and fields that are used are not uniquely identifiable, as is the case with RDF graphs. Nevertheless, this structure allows to describe interlinked data in a coherent way, to query this data and to update or delete it. One of the reasons GraphQL had such an adoption rate over the last years, is the simplicity of querying: essentially, it requires the client to state the desired classes and their fields, possibly accompanied by specific parameters (Table 1). The response will be a JSON object. This simplicity stands in contrast to, for example, SPARQL: although SPARQL is more expressive, a lot of developers prefer the much more minimalistic syntax of GraphQL.

Table 1. Example of a nested GraphQL query and result

Query	**Result**
{	{
architect {	"data":
name	"architect": {
friends {	"name": "Le Corbusier",
name	"friends": [
}	{
}	"name": "Ludwig Mies van der Rohe"
}	},
	{
	"name": "Rem Koolhaas"
	},
	{
	"name": "Andrea Palladio"
	}
	]
	}
	}
	}

Although GraphQL has essentially no relationship to RDF, a GraphQL schema could be provided with the universal context that is missing in the original specification. Such context would extend the schema to the level of the semantic web, using IRIs. Thus far, two open source projects have implemented this extension in a slightly different way: either by implementing the original specification

[7] https://graphql.org/

and schema syntax (HyperGraphQL) or by only adopting the query syntax and translate it to SPARQL 'under-the-hood' (GraphQL-LD).

HyperGraphQL In the approach of HyperGraphQL, an intermediary service is set up, connecting the GraphQL client with the RDF graphs to be queried. Just like in a 'classic' GraphQL approach, HyperGraphQL needs a schema that defines the right types and fields. In RDF terms, the types that are defined relate to the resources defined in an ontology, which means instances of `owl:Class` as well as instances of `owl:Property`. The GraphQL types that refer to an `owl:Class` are then extended by fields that relate to the properties that have this class as a domain, 'pointing' at their range. Types that are not included in the schema, cannot be queried. This provides the data supplier with a mechanism of fine-grained selection of which datatypes can be queried via the 'lens' of a GraphQL schema. A subset of a schema applied to the ICDD, is depicted in Listing 1.1.

```
type __Context {
 InternalDocument:      _@href(iri: "ct:InternalDocument")
 filetype:              _@href(iri: "ct:filetype")
 belongsToContainer:    _@href(iri: "ct:belongsToContainer")
 ContainerDescription:_@href(iri: "ct:ContainerDescription")
}

type InternalDocument @service(id:"icdd") {
 filename: String @service(id:"icdd")
 belongsToContainer: [ContainerDescription] @service(id:"icdd")
}

type filetype @service(id:"icdd") {
}

type belongsToContainer @service(id:"icdd") {
}

type ContainerDescription @service(id:"icdd") {
  [...]
}
```

Listing 1.1. Fragment of a HyperGraphQL schema. <Container> in the Context should be replaced with the original URI.

Apart from that, a HyperGraphQL instance needs a *config.json* file as a setup guide (Listing 1.2. The *config.json* file states data for setting up the server, such as its name, the schema to use and the hosting port. Furthermore, it defines the RDF graphs that can be queried by providing a local path or URI to the graph. Multiple graphs can be included in one configuration file, although types in a schema can only be used to query one service. For example, the types defined in Listing 1.1 are set to be queried against the graph with 'icdd' as a name (defined in the *config.json* file). This means that for querying multiple graphs with

the same types, these should be merged before querying. Listing 1.2 depicts a schema that uses a schema with name *schemaICDD.graphl* (relative path) and queries a local TTL graph with id 'icdd'.

```
{
  "name": "icdd-hgql",
  "schema": "schemaICDD.graphql",
  "server": {
    "port": 8082,
    "graphql": "/graphql",
    "graphiql": "/graphiql"
  },
  "services": [
    {
      "id": "icdd",
      "type": "LocalModelSPARQLService",
      "filepath": "src/main/resources/ICDD.ttl",
      "filetype": "TTL"
    }
  ]
}
```

Listing 1.2. Example of a config.json file used by a HyperGraphQL service.

GraphQL-LD In contrast with the HyperGraphQL approach, GraphQL-LD does not require an additional service to be set up between server and client [11]. Instead, it takes a GraphQL query and expands it with a JSON-LD context. These are then translated into a SPARQL equivalent using a module called 'GraphQL to SPARQL algebra' [8]. The RDF datastore can then be queried using the resulting SPARQL query. A second module, called 'SPARQL results to tree'[9] translates the results of such query to a tree structure that resembles the structure of a 'classic' GraphQL query. GraphQL-LD can also directly be implemented in Javascript applications, due to its integration in the Comunica Framework [10].

Comparison Both solutions have their merits and disadvantages. As they are essentially querying tree structures, they are less expressive than SPARQL, but they allow a more accessible way of querying basic RDF datasets. Nevertheless, neither of them has the option to perform 'mutations', which can alter an existing database. Developers who want their apps to work with existing RDF databases (i.e. creating, reading, updating, deleting), can currently only read data if they use a GraphQL syntax. Furthermore, both are works in progress, which means not all expressivity of GraphQL can be implemented. For example, the concept

[8] https://github.com/rubensworks/graphql-to-sparql.js
[9] https://github.com/rubensworks/sparqljson-to-tree.js

of subclasses is available in GraphQL, but not in HyperGraphQL: querying some fields of a `product:BuildingElement` will not automatically query these fields for subclasses defined in the ontology (`product:Door` or `product:Window` will not be included in the results), because, as far as we know, there is currently no equivalent of GraphQL Union or Interface implemented in HyperGraphQL. In the case of GraphQL-LD, this is less of a problem, since the eventual query is performed using SPARQL. If the query is performed against a SPARQL endpoint with reasoning, there is no real difference with an 'original' SPARQL query. A disadvantage (in certain contexts) of GraphQL-LD, on the other hand, is that it is 'predicate-oriented', which makes it difficult to perform subject-based queries, requiring a workaround that complicates the GraphQL query itself. A more detailed comparison between both query languages has been made by [16], focusing on their potential for distributed services in the building industry.

In the next section, an ICDD container will be queried using these approaches: HyperGraphQL, GraphQL-LD and SPARQL. A GraphQL schema and *config.json* for HyperGraphQL is automatically generated with a basic convertor tool in development[10], as well as a JSON context for GraphQL-LD.

3 Use case: querying an ICDD container

The sample dataset 'usecase_1b_delivery'[11] is used as a case study. It is a lightweight ICDD container that fits for demonstration purposes. Using the abovementioned query languages, the container will be queried for available sources, their name and filetype. Secondly, the container will be searched for links between the documents on a sub-document level. We will compare the results of different query types.

3.1 Querying with HyperGraphQL

In order to set up a query engine with HyperGraphQL, some pre-processing steps need to be carried out first:

1. **config.json**: First, a configuration file is made, containing the information to be used by the HyperGraphQL engine (e.g. the schema that is used and the services that are queried).
2. **Schema conversion**: The next step is the conversion of the ontologies that are present in the ICDD folder into a HyperGraphQL schema. Each type in the schema refers to the service that it needs to query.
3. **Data preparation**: It has been mentioned that types in a HyperGraphQL schema are working on only one service, while a linked ICDD container essentially contains at least two describing graphs: the *index.rdf* graph and the *links.rdf* (or *content.rdf*). Both need to be parsed into a single graph that can be used as one service in the *config.json* file, and to which the types in the schema can refer.

[10] https://github.com/JWerbrouck/Ontology2GraphQL
[11] http://www.iso-icdd.org/examples/part1/usecase_1b_delivery.icdd

4. **Service setup**: Quering with HyperGraphQL requires the setup of an intermediary service. This HyperGraphQL engine is bound to the schema, so if the schema alters, the engine needs to be restarted. After the startup of the service, a graphical interface (*GraphiQL*) for querying the graph is set up, which can be used for performing GraphQL queries.

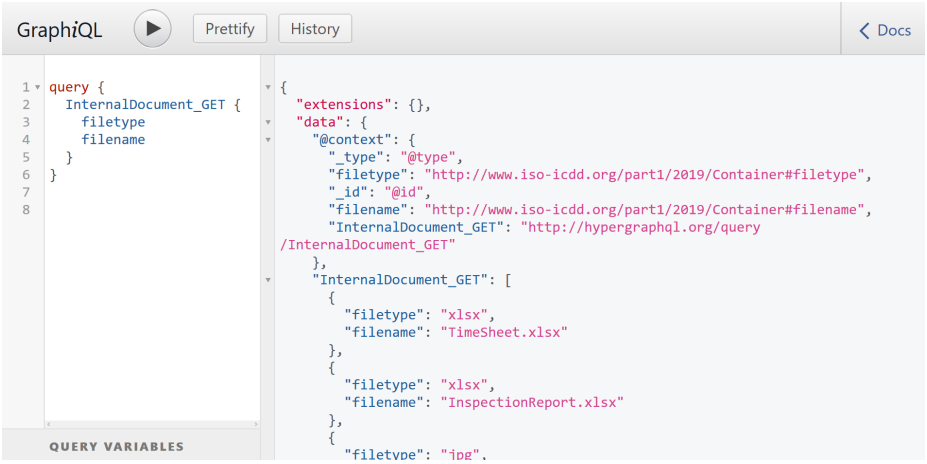

Fig. 3. Querying the (Internal)Documents in an ICDD container

The first query is aimed at finding all documents that are linked in the container (Fig. 3). A first issue is that only subclasses of `ct:Document` are present in the folder. As mentioned, subclass inference is not implemented, so the different types of documents need to be queried explicitly. The second query searches for links between identifiers within documents (Fig. 4). However, the schema had to be adapted manually, for the range of `ls:hasDocument` to focus on `InternalDocument` instead of its superclass `ct:Document`. This is possible because we knew before that there were no other types of documents in the ICDD sample datasets. In a real world situation, this will not always be the case, which makes it quite a large disadvantage. The reason that it was possible in the first scenario to query withouth adapting the scheme, is because there, `InternalDocument` was queried as a GraphQL *type*, not as a *field*.

3.2 Querying with GraphQL-LD

For querying with GraphQL-LD, only a JSON-LD context is needed (e.g. Listing 1.3). Since we focus on querying, we only use the GraphQL-to-SPARQL module and perform the resulting SPARQL query on a triple store (Fuseki) that contains both the *index.rdf* and *links.rdf* graphs. The same query looks slightly different when compared to the query performed in section 3.1, due to the difference

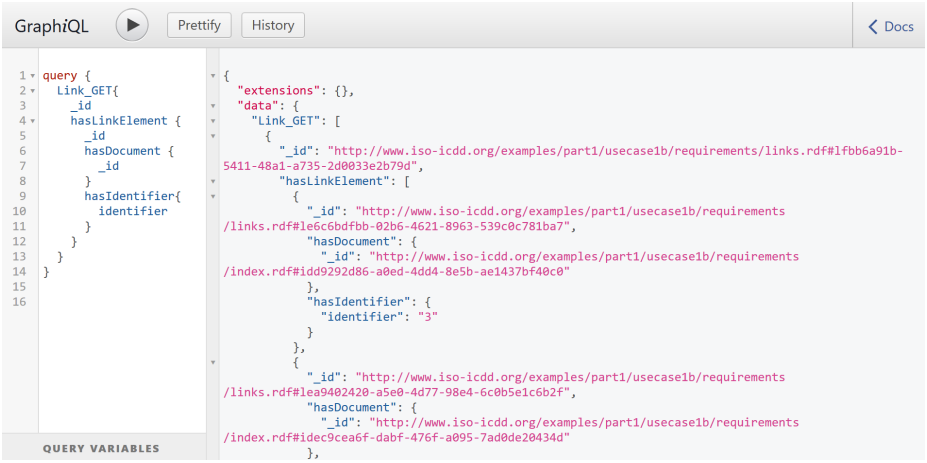

Fig. 4. Querying the links between (internal) documents in an ICDD container

in processing. Because GraphQL-LD is predicate-oriented, a workaround has to be made to query for an object with given class (Listing 1.4). Furthermore, the variables of the query are automatically set to the fields that are queried, ignoring the GraphQL classes. Because there is no field 'uri' (cf. the '_id' field that is assigned to every type in HyperGraphQL), it is therefore difficult to get the URI of a queried type.

The same first query from Section 3.1 (Fig. 3) can be written in GraphQL-LD as depicted in Listing 1.4. This can be queried against any SPARQL endpoint, after conversion by the GraphQL-to-SPARQL module. This yields the SPARQL query that is shown in Listing 1.5.

```
{
"InternalDocument":
    "http://www.iso-icdd.org/[...]/Container#InternalDocument",
"filetype": "http://www.iso-icdd.org/[...]/Container#filetype",
"filename": "http://www.iso-icdd.org/[...]/Container#filename",
"a": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
}
```

Listing 1.3. JSON-LD context for querying with GraphQL-LD

```
{
  a(_:InternalDocument) {
    filetype
    filename
  }
}
```

Listing 1.4. GraphQL-LD query no. 1

```
SELECT ?filetype ?filename WHERE {
  ?b1 a ct:InternalDocument;
    ct:filetype ?filetype;
    ct:filename ?filename.
}
```

Listing 1.5. SPARQL translation of GraphQL query no. 1

Listing 1.6 shows the second query (Fig. 4), which is translated into SPARQL in Listing 1.7. Note that, in this SPARQL translation, only the *document* and the *id* were included as variables. A workaround for this could be to simply query for all variables, by either replacing them with an asterisk manually or by means of a code snippet. To avoid having identical variables, a concatenation takes place to join the variable name with the previous 'branches' in the GraphQL query. A naming of the fields is not required, but can be added for more clarity.

```
{a(_:Link) {
  linkElement: hasLinkElement {
    document: hasDocument
      idUri: hasIdentifier{
        id: identifier
      }
    }
  }
}
```

Listing 1.6. GraphQL-LD query no. 2

```
SELECT ?linkElement_document ?linkElement_idUri_id
WHERE {
  ?b1 rdf:type ls:Link;
    ls:hasLinkElement ?linkElement.
  ?linkElement ls:hasDocument ?linkElement_document;
    ls:hasIdentifier ?linkElement_idUri.
  ?linkElement_idUri ls:identifier ?linkElement_idUri_id.
}
```

Listing 1.7. SPARQL translation of GraphQL query no. 2

3.3 Querying with SPARQL

Finally, the ICDD container is queried using SPARQL. Here, the two queries can be combined into one (1.8). However, in this case the container only has one `ct:ContainerDescription`, which is why this variable is somewhat obsolete and therefore not listed in the results. Furthermore, the query searches for the links, the name and extension of the documents they refer to and the identifiers at sub-document level.

```
PREFIX ct: <http://www.iso-icdd.org/part1/2019/Container#>
PREFIX ls: <http://www.iso-icdd.org/part1/2019/Linkset#>

SELECT ?link ?name ?type ?id
WHERE {
    ?index a ct:ContainerDescription;
        ct:containsDocument ?doc.
    ?doc ct:filename ?name;
        ct:filetype ?type.
    ?link a ls:Link;
        ls:hasLinkElement ?LinkElement.
    ?LinkElement ls:hasDocument ?doc;
        ls:hasIdentifier ?id_guid.
    ?id_guid ls:identifier ?id.
}
```

Listing 1.8. SPARQL translation of GraphQL query no. 2

As was the case in section 3.2, this query can be performed against all SPARQL endpoints, which makes it possible to enable reasoning engines. The results are depicted in Table 2. Three links are present in this ICDD container, two of which connect the IDs within two documents, one of them relates to three IDs.

Table 2. Results of SPARQL query for links between documents on sub-document (identifier) level

link	**name**	**type**	**id**
links:...b79d	Viaduct_48D-100.ifc	ifc	2ptPWWWHjCCfWE1B3yaT2A
links:...b79d	InspectionReport.xlsx	xlsx	3
links:...b79d	damagedConcrete.jpg	jpg	3
links:...2423	Viaduct_48D-100.ifc	ifc	1D_dzaxZf4_QcddQY3uvjg
links:...2423	InspectionReport.xlsx	xlsx	2
links:...e6f09	Viaduct_48D-100.ifc	ifc	20dnvrgv16XurcDlSZ1qI3
links:...e6f09	InspectionReport.xlsx	xlsx	4

4 Conclusion

In this paper, three query syntaxes were discussed to retrieve data from an RDF dataset. The first two rely on the GraphQL syntax, but extend it towards linked data. While HyperGraphQL uses a schema that is fully compliant with the GraphQL specifications, GraphQL-LD translates the syntax of a GraphQL query into SPARQL 'under-the-hood'. Another way for easily querying Linked Data, LDflex, was not included in the discussion because of its dependency on Javascript.

ICDD presents an interesting way of interlinking document-based information with RDF information, providing a way to use RDF to semantically annotate information that is nevertheless stored in a non-Linked Data way. ICDD datasets have a clear structure and are not overly complex, making them suitable to serve as use cases for testing the abovementioned query syntaxes. As ICDD is intended for use within the industry, being able to query it in a simple manner could certainly lower the threshold for developers to use Linked Data in their applications.

It is clear that SPARQL is the most expressive of these three languages. SPARQL also inherently uses context, while for the other two, a context needs to be provided. This makes it difficult to include external data sets on the web, because the statements used in such datasets might not be explicitly available in the context used for querying. However, for simple queries on a known dataset (such as an ICDD container), GraphQL-based queries find their worthiness in the fact that they hide the verbosity of SPARQL behind the elegant and concise syntax of GraphQL. When a fixed dataset is present, such as an ICDD container that contains only internal documents, HyperGraphQL offers a clean way of querying, also due to its 'introspection' feature that helps querying the correct schema definitions. On the other hand, as indicated in [16], the use of GraphQL-LD is preferable in situations in which several endpoints need to be queried, with flexible context adaptation. In other words, when decentralised applications are developed to query distributed datasets. The ICDD ontologies may be of use to link such datasets, owned by different project stakeholders, without the need of creating an ICDD dump file. Querying with HyperGraphQL is somewhat limited in such scenario, since it is more cumbersome to set up and less flexible to incorporate changing query situations. Querying distributed datasets requires a more 'open world' approach that is difficult to implement in HyperGraphQL schemas. It can be argued that the use of fixed HyperGraphQL schemas can be of use to allow fine-grained access to certain building data, based on the role of the person that queries the data, since types that are not present in the schema cannot be queried. However, further research needs to be carried out to gain insight in this potential benefit. Lastly, both GraphQL solutions are essentially only query languages. There are currently no translations to SPARQL CONSTRUCT, or respectively GraphQL mutations. If, in the future, one of these would provide functionality to also update the data using GraphQL syntax, this would open a lot of options for development and querying of basic Linked Data-based services, all behind a clean GraphQL facade.

5 Acknowledgements

The authors would like to acknowledge the support by both the Special Research Fund (BOF) from Ghent University and the BIM4REN project, which has received funding from the European Union's Horizon 2020 Research and Innovation Program under grant agreement No 820773.

References

1. Beetz, J., Van Leeuwen, J., De Vries, B.: Ifcowl: A case of transforming express schemas into ontologies. Ai Edam **23**(1), 89–101 (2009)
2. Beetz, J., Van Leeuwen, J., De Vries, B.: An ontology web language notation of the industry foundation classes. In: Proceedings of the 22nd CIB W78 Conference on Information Technology in Construction. vol. 2006, p. 670. Technical University of Dresden (2005)
3. Facebook Inc.: GraphQL, https://facebook.github.io/graphql/June2018/
4. Gürtler, M., Baumgärtel, K., Scherer, R.J.: Towards a workflow-driven multi-model bim collaboration platform. In: Working Conference on Virtual Enterprises. pp. 235–242. Springer (2015)
5. Pauwels, P., Terkaj, W.: Express to owl for construction industry: Towards a recommendable and usable ifcowl ontology. Autom. Constr **63**, 100–133 (2016)
6. Rasmussen, M., Pauwels, P., Lefrançois, M., Schneider, G., Hviid, C., Karlshøj, J.: Recent changes in the building topology ontology. In: LDAC2017-5th Linked Data in Architecture and Construction Workshop (2017)
7. Rasmussen, M.H., Pauwels, P., Hviid, C.A., Karlshøj, J.: Proposing a central aec ontology that allows for domain specific extensions. In: 2017 Lean and Computing in Construction Congress (2017)
8. Schapke, S.E., Katranuschkov, P., Scherer, R.J.: Towards ontology-based management of distributed multi-model project spaces. In: Proceedings of the CIB-W78 conference on IT in construction (2010)
9. Semantic Integration Ltd.: HyperGraphQL, https://www.hypergraphql.org/
10. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a modular sparql query engine for the web. In: International Semantic Web Conference. pp. 239–255. Springer (2018)
11. Taelman, R., Vander Sande, M., Verborgh, R.: GraphQL-LD: Linked Data Querying with GraphQL. In: ISWC2018, the 17th International Semantic Web Conference (2018)
12. Taelman, R., Vander Sande, M., Verborgh, R.: Bridges between graphql and rdf. In: W3C Workshop on Web Standardization for Graph Data. W3C (2019)
13. Törmä, S.: Semantic linking of building information models. In: 2013 IEEE Seventh International Conference on Semantic Computing. pp. 412–419. IEEE (2013)
14. Verborgh, R.: Designing a linked data developer experience (2018)
15. Werbrouck, J., Pauwels, P., Beetz, J., van Berlo, L.: Towards a decentralised common data environment using linked building data within the solid ecosystem. In: Proceedings of the 36th CIB W78 Conference on Information Technology in Construction. CIB W78 (Under review)
16. Werbrouck, J., Senthilvel, M., Beetz, J., Bourreau, P., van Berlo, L.: Semantic query languages for knowledge-based web services in a construction context. In: 26th International Workshop on Intelligent Computing in Engineering. EG-ICE (Under review)