Faculty of Computer Science

Master Thesis

Querying Wikidata with GraphQL

Anas Shahab

Master Computational Logic

Matriculation number: 4827407

Supervisor:

Prof. Dr. Markus Krötzsch

Second Reviewer:

Dr. Dörthe Arndt

Tutor:

Dipl.-Inf. Lukas Gerlach

Faculty of Computer Science

Institute of Theoretical Computer Science

Chair of Knowledge-Based Systems

Submission Date: XX.XX.2023

Declaration of originality

I hereby declare that I have written this Thesis on my own accord and any participation of others has been acknowledged. I have clearly marked all references to existing work. I have not submitted this work partly or as a whole anywhere else.

Dresden, XX.XX.2023

_____

(signature)

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

Chapter 1

Introduction

The term "knowledge graph" gained popularity in 2012 when Google launched its own *Google Knowledge Graph*. A knowledge graph is a collection of data represented as a graph. The collected data conveys knowledge of the real world, where the nodes represent entities of interest and edges the many different relations between those entities [1]. The entities are real world objects and abstract concepts. For example, "Helium has the chemical formula He", is a statement that can be represented using a knowledge graph. Here the nodes of the graph would represent "Helium" and "He", while the connecting edge between those nodes would represent the relation "chemical formula".

There are many ways of modelling data as a graph. The most commonly used ones are directed edge-labelled graphs, heterogeneous graphs, property graphs and graph dataset [1]. We will see in Section 2 how we can use

the? (I get why you did not put it though)

Resource Description Framework (RDF)[1] to specify directed edge-labelled graphs.

Many companies such as Amazon, Facebook, Uber, Google, etc., use knowledge graphs for their applications. Depending on the organization or community there are open or enterprise knowledge graphs [1]. Open knowledge graphs include Wikidata, DBpedia, Freebase, YAGO, etc. These are available online and freely accessible to the public. Enterprise knowledge graphs are generally used internally within a company and have their commercial specific use-cases [1].

do not put this citation after every sentence; maybe after a paragraph

[1] https://www.w3.org/TR/rdf11-concepts/ .

Wikidata is an open knowledge graph developed by Wikimedia Deutschland. It contains structured data and partly acts as a central database for Wikimedia projects like Wikipedia. Wikidata is built on RDF framework. This enables it to be queried using SPARQL, which is a query language for RDF. However, in commercial applications the use of SPARQL remains limited. One of the main reasons is that developers are not often learned or experienced in the triples that RDF offers, and find the structure of SPARQL queries to be complex.

this claim cannot be backed by any evidence; this is only speculation and can be phrased as such. Certainly SPARQL is quite complex compared to other query formalism but we don't know if this is the reason for it not being used widely (but it might be). Also I'm not even sure that SPARQL is not used widely. Maybe many applications actually use it.

GraphQL is a query language popular in commercial applications. It was developed by Facebook in 2012 and made open source in 2015. It is easy to learn and use, providing syntax that is more human friendly than SPARQL. Our goal in this report is to provide a research on ways to query Wikidata using GraphQL, and offer two implementations for this purpose - GraphQL-LD and HyperGraphQL. Both of these are open source and can be used to query arbitrary knowledge graphs using GraphQL.. We also provide comparisons between the implementations, and evaluate their performances along with limitations.

this might need to change depending on what will really be part of the evaluation chapter

motivation could still be clearer I think

The remainder of the report is structured as follows.

- In chapter 2, we provide an overview of RDF, Wikidata, SPARQL and GraphQL. We also give a comparison between GraphQL and SPARQL.
- In chapter 3, we show the approaches used to query RDF graphs using GraphQL.
- In chapter 4, we aim to provide the implementation of the above approaches on Wikidata. This also contains the technicalities and the differences in the SPARQL queries generated by the tools. Moreover, we also show the differences between the generated SPARQL queries and handwritten SPARQL queries here. Finally, we end the chapterby providing the performance and limitations of the tools.

increase introduction

Chapter 2

Preliminaries

In this chapter, we introduce some basic concepts of RDF, Wikidata, SPARQL, and GraphQL. Then, we discuss the differences between SPARQL and GraphQL in terms of ease of use by developers in their applications and expressibility.

2.1 RDF

The World Wide Web consists of data published in various formats such as PDF, CSV and many forms of plain text [4]. Linked Data turns the web into a global database where data can be reused and shared across to everybody. Resource Description Framework (RDF) is a framework used to represent information available in the Web [5]. In the context of graphs, RDF is used for describing and exchanging graphs. The graphs specified by RDF are directed edge-labelled graphs. This means that the edges connect source nodes to target nodes, and have labels. It can be the case that there are multiple edges between the same nodes. However, these edges must have different labels. Figure 2.1 shows how knowledge about the chemical element Helium can be represented using a directed edge-labelled graph.

RDF Schema (RDFS) is the Vocabulary Description Language for RDF. Basically, it defines the vocabulary for RDF data. This means that it describes:

- the basic concepts and abstract syntax of RDF such as resources and classes[2]
- the formal semantics of RDF[3]
- the different concrete syntaxes such as Triples, which is shown in section XYZ

[2]https://www.w3.org/TR/rdf-concepts/ .
[3]https://www.w3.org/TR/2014/REC-rdf11-mt-20140225/ .

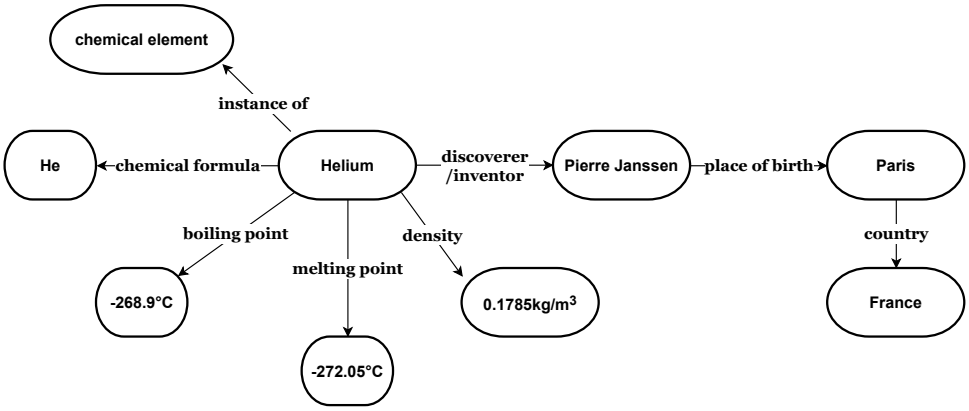

Figure 2.1: Directed edge-labelled graph describing Helium

According to RDFS, resources are divided into groups known as classes. Each member of a class is called an instance of that class, and is itself also a resource. The relationship between subject and object resources is described via RDF properties Essentially, the predicate of an RDF statement is an instance of RDF property. For instance, to identify that a resources is an instance of a class we use the predicate rdf:type which is in turn an instance of RDF property. W3cC provides a thorough documentation on RDF Schema.[4] In Figure 1 we see that Helium is an instance of the class chemical element. The property instance of describes the relation between the subject Helium and object chemical element.

Is the concept of property and predicate clear?

honestly, I don't quite get it yet; let's discuss

In order to exchange graphs across the web we need to identify the resources uniquely. For this we use IRIs which are basically identifiers in RDF. The graph shown in Figure 2.1 can be represented using an RDF graph. Formally, the building blocks of RDF graphs are IRIs, literals and blank nodes. These are defined as follows.

IRI

A Uniform Resource Identifier (URI) is a sequence of a subset of ASCII characters that identifies any web resource by using a name, a location, or both. They have a scheme, authority, path, and query and fragment, where all parts other than scheme and path are optional. URIs are of the form **scheme:[//authority]path[?query][#fragment]**. For example, *http://www.wikidata.org/entity/Q560* is an IRI that identifies the chemical element Helium on Wikidata. A Uniform Resource Locator (URL) is a subset of URI that is used to specify the location of a digital document.

[4] w3.org/TR/2014/REC-rdf-schema-20140225/ .

An Internationalized Resource Identifier (IRI) is a generalized form of URI that helps to distinguish resources with Unicode. Basically, the character set in URI is extended to the Universal Coded Character Set. This enables it to contain any Latin and non-Latin characters except the reserved characters.

In RDF an IRI is used as a name (can be thought of as an ID) for graph nodes. It defines the resources that appears as nodes or edge labels in a RDF graph. There are already several pre-existing IRIs available for common use. New domain specific IRIs can be created based on the application. However, we must ensure there no conflicts with other IRIs available on the web.

RDF Literals

An RDF literal consists of three essential elements: a lexical value, a datatype IRI and an optional language tag. The lexical value is a string[5] that corresponds to a particular literal value in the value space, where value space is the set of all possible values that a datatype can have. There are many datatypes[6] in RDF some of which are string, Boolean, decimal and integer.

The datatype IRI refers to a datatype that defines which strings are valid (belong in the lexical space), the value space and the lexical-to-value mapping [6]. This mapping is essentially a function that maps each string from the lexical space to an element in the value space. The W3C standard XML Schema defines the datatypes and their IRIs. For example, decimals are identified by the IRI http://www.w3.org/2001/XMLSchema#decimal. W3C has a good documentation on the different XML Schema built-in datatypes [5].

The optional language tag helps to provide human-readable labels to RDF literals. A literal is a language-tagged string is of the form "string"@language.[7] The datatype IRI[8] of such literals is http://www.w3.org/1999/02/22-rdf-syntax-ns#langString.

RDF literals are used to represent resources that have values belonging to datatypes. Each literal can have only one datatype. For example, the boiling point of Helium would be a RDF literal represented as `-268.98sd:decimal` and its chemical formula as `"He"@en`, which is a language-tagged string. Literals are drawn as rectangular nodes in RDF graphs.

[5]RDF is based on Unicode strings.

[6]A full list is available on the W3C's section on RDF datatypes: www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#section-Datatypes.

[7]Here language is a well-formed language tag (after BCP47).

[8]It is never used in syntax.

Blank Nodes

A blank node in RDF, also known as a bnode, does not identify a specific resource as IRIs or literals do. It is used as a placeholder for some node, i.e., it is used to say that something with the given relationship exits at the position without specifying what the node is.

2.1.1 RDF Graph

Definition 2.1.1 (RDF Graph)**.** An RDF graph is a directed edge-labelled graph composed of a set of triples. A triple, also known as statement, represents the relationship between a subject and an object, linked by a predicate as shown in Figure 2.2. Formally, each triple consist of the following elements:

- a subject node that is an IRI or a blank node
- a predicate edge that is an IRI
- an object node that is an IRI, a blank node, or a literal

Figure 2.2: An RDF graph with Subject and Object nodes connected via a predicate edge

Figure 2.3 shows an RDF graph based on our example represented in Figure 2.1. Our main interest is in querying the knowledge graph Wikidata, and so all the data correspond to the resources in its knowledge base. In Wikidata the subject and object represent items, and the predicate represents properties. All items and properties are identified as Unique IDs. For example, the item *Helium* has the ID of `Q560`, and the property *chemical formula* has the ID `P274`. These are not understood by humans and have a label property that makes them understood. Moreover, items belong to the namespace `http://www.wikidata.org/entity/` (prefixed by `wd`) and properties to `http://www.wikidata.org/prop/direct/` (prefixed by `wdt`). As a result, *Helium* would have the IRI `http://www.wikidata.org/entity/Q560` (`wd:Q560`) and *chemical formula* the IRI `http://www.wikidata.org/prop/direct/P274` (`wdt:P274`). Section XYZ gives an elaborate understanding of entities and the namespaces they belong to in Wikidata.

From Figure 2.3 we understand that *Helium* (`Q560`) is an *instance of* (`P31`) of *chemical element* (`Q11344`). It has a human understandable english *label* called *helium* and the *chemical*

formula (P274) of *He*. Its *boiling point* (P2102), *melting point* (P2102) and *density* (P2054) are *-268.9 °C*, *0.1785 °C* and *-272-05 kg/m3* respectively. Helium has a *discoverer/inventor* (P61) by the name of *Pierre Janssen* (Q298581). His *place of birth* (P19) was *Paris* (Q90) that belongs to the *country* (P17) of *France* (Q142).

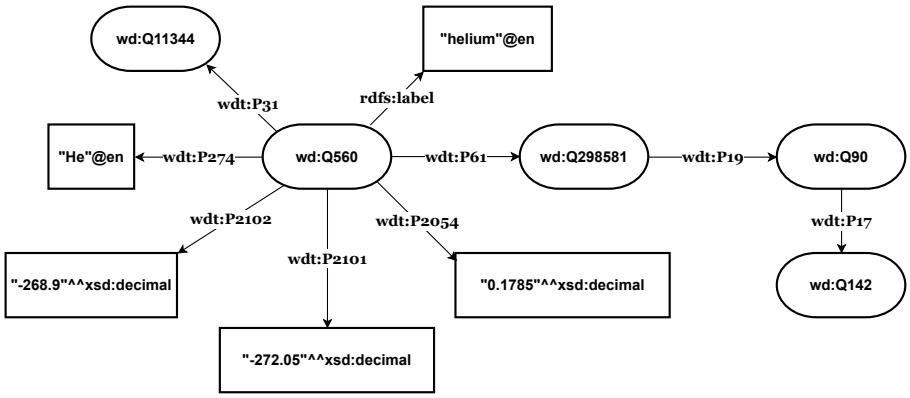

Figure 2.3: RDF graph describing Helium

2.1.2 Serialisations

For exchanging graphs across the web, we need a syntactical representation of RDF. There are different formats available, the most common ones are N-Triples, Turtle, JSON-LD, RDF/XML and RDFa. In this report we focus on N-Triples and Turtle.

N-Triples

N-Triples represents RDF graphs in a simple line-based format.[9] Every triple is encoded in a single line. The IRIs are written within pointy brackets and literals are written as lexical value^^datatype-IRI. Blank nodes are represented as _:stringID, where stringID can be any string used to identify the blank node in the document. After every element of a triple there is a whitespace, and all the lines end with a dot. We can use comments using hash symbol after the end of every triple in a line or in a single dedicated line, and they are treated as white spaces. The files are saved with a *.nt* extension.

Listing 2.1 shows the representation of the RDF graph in Figure 2.3 in N-triples format. We have given line breaks for better readability.

[9]Full specification available at: https://www.w3.org/TR/n-triples/ .

```
<http://www.wikidata.org/entity/Q560> <http://www.wikidata.org/prop/direct/P31>
                                              <http://www.wikidata.org/entity/Q11344> .

<http://www.wikidata.org/entity/Q560> <http://www.w3.org/2000/01/rdf-schema#label>
                                              "helium"@en .

<http://www.wikidata.org/entity/Q560> <http://www.wikidata.org/prop/direct/P274>
                                              "He"@en .

<http://www.wikidata.org/entity/Q560>  <http://www.wikidata.org/prop/direct/P2102>
                                              "-268.9"^^<http://www.w3.org/2001/XMLSchema#decimal> .

<http://www.wikidata.org/entity/Q560> <http://www.wikidata.org/prop/direct/P2101>
                                              "0.1785"^^<http://www.w3.org/2001/XMLSchema#decimal> .

<http://www.wikidata.org/entity/Q560> <http://www.wikidata.org/prop/direct/P2054>
                                              "-272.05"^^<http://www.w3.org/2001/XMLSchema#decimal> .

<http://www.wikidata.org/entity/Q560> <http://www.wikidata.org/prop/direct/P61>
                                              <http://www.wikidata.org/entity/Q298581> .

<http://www.wikidata.org/entity/Q298581> <http://www.wikidata.org/prop/direct/P19>
                                              <http://www.wikidata.org/entity/Q90> .

<http://www.wikidata.org/entity/Q90> <http://www.wikidata.org/prop/direct/P17>
                                              <http://www.wikidata.org/entity/Q142> .
```

Listing 2.1: RDF graph represented in N-triples syntax

Turtle

Turtle is an easy to read representation of RDF graphs. It extends the N-Triples format by providing several simplifications.[10] We can use prefix declarations and base namespaces at the beginning of the file to shorten IRIs. Turtle allows us to avoid repetition. We can use a semicolon at the end of a line instead of a dot if we know the next line has the same subject. Consequently, the next line will only have a predicate and object omitting the subject. Also, we can use a comma at the end of a line if we know the next line will start with the same subject and predicate. Analogously, the next line will only have a an object omitting the subject and the predicate. Blank nodes are represented using only square brackets. Additionally, we can provide predicate-object pairs within the square brackets to give further triples keeping the blank node as the subject. Turtle also provides a shorthand syntax for writing numbers. Numbers of the datatype integer, decimal and double can be written without quotes and datatype-IRIs. Boolean values can be written directly as either "*true*" or "*false*". The files are saved with a *.ttl* extension.

Listing 2.2 shows the representation of the RDF graph in Figure 2.3 in Turtle format.

[10]Full specification available at: https://www.w3.org/TR/turtle/ .

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

wd:Q560 wdt:P31 wd:Q11344 ;
        rdfs:label "helium"@en ;
        wdt:P274 "He"@en ;
        wdt:P2102 -268.9 ;
        wdt:P2101 0.1785 ;
        wdt:P2054 -272.05 ;
        wdt:P61 wd:Q298581 .
wd:Q298581 wdt:P19 wd:Q90 .
wd:Q90 wdt:P17 wd:Q142 .
```

Listing 2.2: RDF graph represented in Turtle syntax

2.2 Wikidata

Wikidata is a free and publicly available knowledge base that can be read and edited by both humans and machines. It is one of the many projects by Wikimedia Foundations such as Wikipedia, Wikibooks, Wikimedia Commons and Wikitionary. Wikidata was created in 2012 at Wikimedia Deutschland by a community of volunteers. These volunteers edit and control all content. As of December 2022, Wikidata has more 23000 active editors.[11] Wikidata provides a website where data can be viewed and also edited [7].

One of the original purposed behind the creation of Wikidata was to help its sister projects. Initially, Wikipedia and its sister projects used to maintain their own lists of interlanguage links. This refers to links between Wikipedia articles about the same topic but in different languages. After 2012, these interlanguage links were provisioned via Wikidata. Wikidata is also used to display data shown inside the pages in Wikipedia. The usage of this mainly depends on the language of the Wiki. For instance, in the case of some languages, parts of the Wikipedia pages are created automatically from the data in Wikidata. Others, especially those of smaller languages that are not widely used, use Wikidata to create placeholder pages when an article may not be in their respective language. In 2018, around 59% of Wikidata information was used in English Wikipedia articles, although mostly for external identifiers and coordinate locations [**?**].

Wikidata stores information in the form of structured data in a database [8]. This is not the case for its sister projects as they contain unstructured data stored. The information on their web pages is not directly given a structure in the form of tables or lists. Wikidata acts as a central

[11] https://wikidata.wikiscan.org/ .

storage for these projects and focuses on providing a structure for their data [**?**].

typo

Addtionally, Wikidata also supports linked data. This means that the data stored can be linked to datasets and databases like Google Books and OmegaWiki. Wikidata can also be used for quality checks against Wikipedia articles. This is useful when information about a specific topic needs to be known and the solution is easily found by querying the knowledge graph, in this case Wikidata.

Moreover, there is also a wide range of commercial and research oriented applications for Wikidata. This is due to the fact that it has a large amount of real world data. For instance, Wikidata has external usages in many large organizations such as Eurowings, Google, Apple and Amazon. This includes tasks such as data integration, authority control, identity providing and data-driven journalism. In the field of research, Wikidata is used for collecting test data for knowledge graph related algorithms and training data for machine learning projects.

Wikidata is built on the RDF framework. However, it does not define its resources in terms of RDF. Instead, it has its own model known as Wikibase data model.[12] This distinction creates an abstract layer between its model and RDF. Consequently, there are similarities with RDF's W3C standards but there are also some important differences. For instance, the Wikidata property *instance of* (`P31`) is semantically equivalent to the property *rdf:type* in RDF. The value of `P31` is a class that is itself an item. Wikidata offers an explanation for some of the standards properties in Wikidata that correspond to the ones in RDF [9]. Statements (triples) in Wikidata can have annotations (qualifiers) and references.

The basic elements in Wikidata are entities, also known as resources. These are mainly items and properties. The next section describes entities in Wikidata.

2.2.1 Entities

Wikidata maintains its data structure by pages. Each entity has a dedicated page for itself. Basically, every element on which Wikidata has structured data is known as an entity[10]. Entities are identified by an unique ID and not by names or labels. Currently, Wikidata has mainly three types of entities – item, property and lexeme. Other extensions can define new entity types such as MediaInfo and subentities like Form and Sense. We will discuss about items and properties. The rest are out of the scope of this report.

Items are real-world objects, concepts or events. According to RDF terminology, items are instances and classes. Instead of a human understandable name, they are identified by a QID -

[12]https://www.mediawiki.org/wiki/Wikibase/DataModel.

an ID prefixed with the letter "Q" and followed a number. Items in Wikidata belong to the main namespace - *http://www.wikidata.org/wiki/QID*. Every item constitutes of the following main parts - labels, descriptions, aliases, sitelinks and statements[10].

Labels, descriptions and aliases are multilingual which help to find the respective item. Since items are identified by an ID, these are used to identify the items clearly. Sitelinks provide links about an individual item in Wikidata to external pages on other Wikimedia sites such as Wikipedia and its other sister projects. The most important part are statements.

A statement in Wikidata consists of a claim, references and a rank. Claims are property-value pairs, which along with the item form a RDF triple, where the item is the subject. Claims can also contain optional qualifiers that provide some additional information for the claim. This information is a property-value pair that refers to the main part of the statement instead of the item itself[10]. References point to resources that support the claim. An item can have several statements for the same property, and all of them might not necessarily be important or relevant. Ranks are used as a quality factor to distinguish between several statements. A Wikidata statement can have one of three types of ranks - "normal", "preferred" and "deprecated". By default, a statement has the normal rank unless changed to preferred or deprecated. A statement with preferred rank means it should be given priority over the normal ranked statements. A deprecated rank indicates that the statement is incorrect or under discussion, and it may have a reference. Deprecated statements are kept either for the sake for completion or to prevent users from constantly adding or removing them. Wikidata has around 100 million items[13] and around 1.43 billion item statements as of December 2022.[14]

Figure 2.4 shows an excerpt of the Wikidata page on the item Helium.[15] We can see Helium has the QID of Q560, and has labels, descriptions and aliases in different languages. It has a sitelink for the item in Wikipedia offered in 186 languages. Helium has a statement that indicates that Helium has the instance of property with chemical element as the values. The property-value pairs follows, hydrogen and followed by, lithium are qualifiers. This statement hence gives us the information that Helium is an instance of chemical element, and it comes after the element hydrogen and if followed by the element lithium. This statement has no references and has the normal rank (indicated by the middle portion greyed).

Properties in Wikidata resemble RDF properties and are essentially attributes for describing entities. They are identified by a PID - an ID prefixed with the letter "P" and followed a number. They belong to the property namespace in wikidata - http://www.wikidata.org/wiki/Property:PID. Like items, they also have labels, descriptions,

[13]https://grafana.wikimedia.org/d/000000167/wikidata-datamodel.
[14]https://grafana.wikimedia.org/d/000000175/wikidata-datamodel-statements.
[15]https://www.wikidata.org/wiki/Q560.

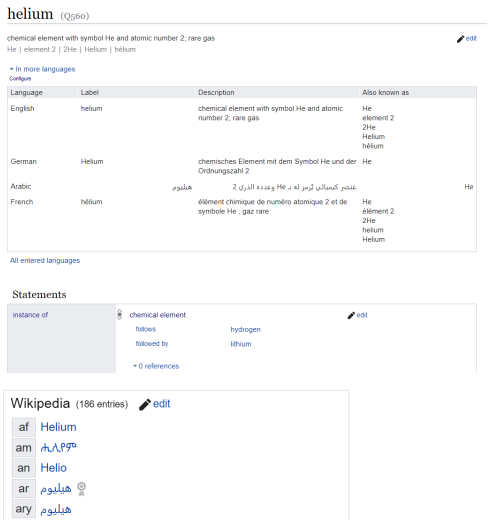

Figure 2.4: An excerpt of the page on *Helium* in Wikidata

aliases and statements but no sitelinks. However, they has an additional part called datatype that determines which values they accept, such as string, quantity and time.[16]

Figure 2.5 shows an excerpt of the property instance of page on Wikidata.[17] The property has a PID of P31 along and with multilingual labels, descriptions and aliases. From the statement we understand that it this property also has an instance of property with Wikidata property being the value. The statement has no qualifiers or references, and has the normal rank. The instance of property accepts the datatype item as a value.

2.2.2 Querying Wikidata

The easiest and most popular way to query Wikidata is through the Wikidata Query Service (WDQS). This is Wikidata's SPARQL endpoint. We can use this service two ways. Firstly, we can write queries in SPARQL directly on the web user interface of the service[18] and obtain the results in different formats like table, tree, graph, etc. Secondly, the service can also be used progmatically by submitting GET or POST requests.[19]

Another popular way to query Wikidata is by using the Wikidata API.[20] However, this API should mainly be used when we want to edit the contents of Wikidata or get data about entities like revision history.

Wikidata dumps is useful when we know our result set will be significantly large or if we want

[16]Wikidata provides a list of all the datatypes: https://www.wikidata.org/wiki/Special:ListDatatypes.
[17]https://www.wikidata.org/wiki/P31.
[18]https://query.wikidata.org/ .
[19]https://query.wikidata.org/sparql.
[20]https://www.wikidata.org/wiki/Special:ApiSandbox.

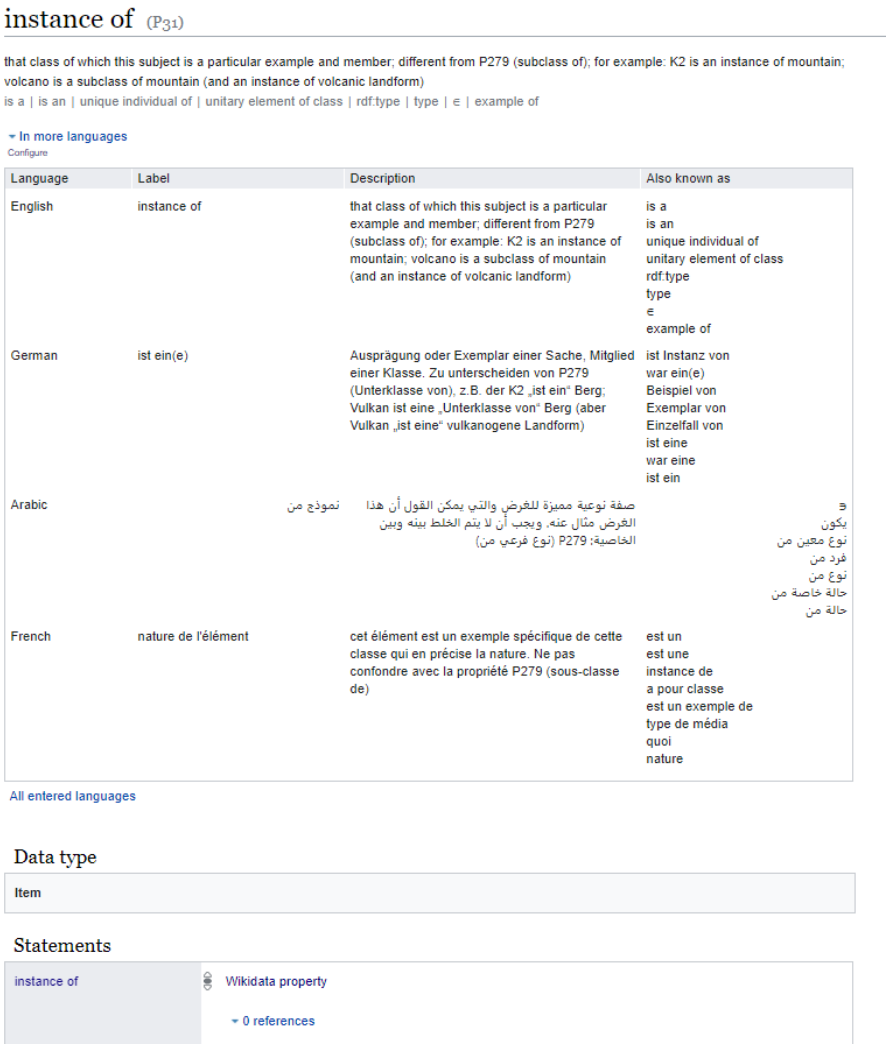

Figure 2.5: An excerpt of the page on *instance of* in Wikidata

to set up our own local query service. These dumps are full exports of all the available entities in Wikidata.[21] To get started you should download the latest complete dump.[22] Wikidata also mentions some other ways to accessing Wikidata's data like Search and Linked Data Fragments endpoint, the complete list and usage of which can be found on Wikidata's Data Access webpage [11].

2.3 SPARQL

SPARQL Protocol and RDF Query Language (SPARQL)[23] is a W3C recommended query language for RDF. This means it allows to query any data source that can be mapped to RDF. It is

[21] https://dumps.wikimedia.org/ .
[22] https://dumps.wikimedia.org/wikidatawiki/latest/ .
[23] https://www.w3.org/TR/sparql11-overview/ .

also a HTTP-based protocol for linked open data on the web. This enables the transmission of SPARQL queries and results between a client and a SPARQL engine. The first working draft for SPARQL was released in 2004 and it became a W3C Recommendation in 2008 [12].

Queries in SPARQL are based on matching graph patterns and can be used to retrieve, add or delete data in the RDF based dataset. In section XYZ we saw that RDF data is based on triples - subject, object and predicate. Consequently, a query in SPARQL consists of a set of triple patterns. Each of the elements of the triple can be a variable (a string beginning with ? or $) that needs to be queried. The solution to the variables is obtained by matching the query patterns to the triples in the dataset.

There are four forms of queries - SELECT, ASK, CONSTRUCT and DESCRIBE.

- SELECT queries select some or all the pattern matches and provides the results in a tabular format
- ASK queries check whether there is at least one match and the result is true or false
- CONSTRUCT queries create an RDF graph based on the query results
- DESCRIBE queries return a RDF graph providing additional information on each results

In our work, we only consider SELECT queries. These consist of the following major blocks:

- Prologue: PREFIX and BASE keywords that function similarly to those in RDF turtle format
- Select clause: SELECT keyword followed by either a list of variables and variable assignments, or by *
- Where clause: WHERE keyword followed by a query graph pattern to be matched
- Solution set modifiers: Change the set of solutions using modifiers such as LIMIT and OFFSET

The select and where clauses are mandatory, the rest being optional. Other optional features are filters, groups, query operators such as UNION, OPTIONAL and BIND, and aggregates. A full specification for the query language can be found on the official W3C documentation[**?**].

Listing 2.3 shows an example of querying Wikidata using SPARQL. We want to get a a list of all chemical elements, along with their English labels, that have a chemical formula, boiling point, melting point, density, an inventor/discoverer, birth place of that inventor/discoverer and the country that the place belongs to. Since there might be several results, we are limiting them to five using the LIMIT keyword. The namespace http://www.wikidata.org/entity/ is used for

items when querying. We are interested in the truthy values of the properties and so the namespace http://www.wikidata.org/prop/direct/ is used for properties. Truthy values are essentially the values for which the statement has the best non-deprecated rank. This means that if a statement has preferred rank then that statement is considered to be truthy. Otherwise, the normal ranked statement is taken to be truthy. The PREFIX is optional since Wikidata recognizes the short forms wd and wdt automatically. Turtle syntax that we saw in section XYZ can be applied in SPARQL. The query can be run in Wikidata's query service.[24]

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT *
WHERE {
  ?element wdt:P31 wd:Q11344 ;
           wdt:P274 ?element_formula ;
           wdt:P2102 ?boiling_point ;
           wdt:P2101 ?melting_point ;
           wdt:P2054 ?density ;
           wdt:P61 ?discoverer .
  ?discoverer wdt:P19 ?place_birth .
  ?place_birth wdt:P17 ?country .
  FILTER(LANG(?element_label)="en")
}LIMIT 5
```

Listing 2.3: Querying Wikidata with SPARQL

Table 1 shows the results obtained in a tabular form. Among the results, there is the element Helium (Q560) that we have used as an example in Fig 1 and Fig 3.

(fix table)

Table 2.1: Results of the SPARQL query in Listing 2

element	element_formula	element_label	boiling_point	melting_point	density	discoverer
wd:Q560	He	helium	-268.9	-272.05	0.1785	wd:Q298581
wd:Q560	He	helium	-268.9	-272.05	0.1785	wd:Q950726
wd:Q560	He	helium	-268.9	-272.05	0.1785	wd:Q127959
wd:Q670	Si	silicon	4271	2570	2.329	wd:Q151911
wd:Q568	Li	lithium	1317	180.5	0.535	wd:Q313568

2.4 GraphQL

GraphQL (Graph Query Language) is an open source query language for APIs (Application Programming Interfaces) and a runtime for executing those queries against existing data. It

[24]https://query.wikidata.org/ .

describes data structured in a graph format – a collection of objects (nodes) connected to each other by some kind of relationships (edges). Runtime is usually implemented by a server.

GraphQL is usually served over HTTP through a GraphQL server. A GraphQl server consists of two main parts – schema and resolver. The API developers create a schema that is strictly typed and describes all the possible data a client can query using the service. The schema specifies object types and fields along with operations on those types. The object type represents the kind of object that can be requested by a client.

There are three operations types – query, mutation and subscription. We look at the query operation in this chapter; the other two are outside the scope of this report. Queries are used to fetch or read data. When compared to REST (Representational State Transfer), queries operations in GraphQL are similar to GET requests. A resolver in GraphQL server is a function that is associated with every field and contains instructions on how to process that particular field. In other words, the resolver is responsible for retrieving a value from the data source.

GraphQL is data agnostic, i.e., it is not concerned where the data source is located. The data could be stored in any source such as a database or a micro-service as shown in Figure XYZ. With a single API call, GraphQL can aggregate data from multiple sources and resolve the data to the client. This is one of the advantages GraphQL has over REST API where the latter would require several HTTP calls to access data from multiple sources. Apart from being data agnostic, GraphQL is also language agnostic. This means that GraphQL services, such as the schema and resolvers, can be written in any programming language such as JavaScript or Python.

For the sake of human readability, GraphQL specification has its own Schema Definition Language (SDL). It is simple to write and understand schemas in SDL, and is similar to the language that we use to write queries. Listing XYZ shows a schema written in SDL. This schema can be in the GraphQL server against which clients can send queries for instances of chemical elements that have a name, chemical formula and boiling point. The exclamation mark means that the corresponding field is non-nullable and it is expected that GraphQL will give a value when the field is queried. A complete guide on schemas and types can be found on the official documentation from GraphQL.[25]

[25]https://graphql.org/learn/schema/ .

```
type Query{
  chemicalElement: ChemicalElement
}

type ChemicalElement{
  name: String!
  chemicalFormula: String!
  boilingPoint: Float!
}
```

Listing XYZ shows a query that can be used against this schema and the results that could be obtained. The results obtained are in JSON format. The official GraphQL website provides a comprehensive documentation on querying a GraphQL server. [26]

```
query QueryChemicalElement (limit 2){
    chemicalElement{
    name
    chemicalFormula
    boilingPoint
  }
}
```

Listing 2.4: Query to fetch chemical elements and their properties

```
{
  "data":{
    "chemicalElement":{
      "name": "helium",
      "chemicalFormula": "He",
      "boilingPoint": -268.9
    },
    {
      "name": "silicon",
      "chemicalFormula": "Si",
      "boilingPoint": 4271
    }
  }
}
```

Listing 2.5: Query to fetch chemical elements and their properties

2.5 GraphQL vs SPARQL

GraphQL and SPARQL are query languages developed with different goals in mind. GraphQL was designed mainly to wrap REST APIs in a graph like shape. This would allow fetching related data in a single request with the aid of schemas. In REST API, the same would require calling multiple endpoints. SPARQL, on the other hand, was developed mainly as a query language for RDF graphs.

[26] https://graphql.org/learn/queries/ .

SPARQL is immensely popular in the field of research and academia. However, it has not seen much growth in commercial applications. GraphQL, on the other hand, is more popular among software and web developers. There are some valid reasons for this.

Firstly, many developers are still not familiar with the Linked Data model of RDF and SPARQL. They are more used to working with technologies such as GraphQL and RESTAPIs. Secondly, GraphQL is simple to learn and work with since it has human-oriented syntax. Among other things, this benefits application development. SPARQL however, proves to be more complex. It has more syntactic verbosity owing to the descriptive nature of writing queries. The produced output from SPARQL queries contains a lot of unnecessary metadata that is not useful for web developers and need to be further parsed for using in web applications [13].

Moreover, retrieval of data from knowledge graphs using SPARQL is time consuming and proves to be a steep learning curve, the reason being that there is limited documentation available for proper ontology descriptions and examples of using queries for SPARQL endpoints [14]. One of the other reasons for using GraphQL over SPARQL is that developers are more equipped in using nested objects that GraphQL offers [15]. They lack the experience to work with triples which is the main foundation of SPARQL. Moreover, fewer supporting tools like libraries and frameworks exist for working and developing with SPARQL than with GraphQL [15].

On the other hand, SPARQL also has some advantages over GraphQL. SPARQL queries represent full graphs while those of GraphQL represent trees [15]. This make SPARQL more expressive. RDF provides a system to build detailed structures from the meaning of data, and is hence more complete and capable than GraphQL schemas [16]. Since SPARQL works with the schema organization of RDF, this makes it more powerful than GraphQL. With SPARQL we can write complex queries that can be used to retrieve or modify data [14]. As a result it is capable to satisfy many complex use cases.

Also since GraphQL alone has no notion of semantics [15], a schema needs to be defined by the GraphQL API developers for every interface they want the client to query. This makes it difficult to integrate data returned when querying multiple different sources. SPARQL however supports federated queries which makes it more powerful and rich. Lastly, when using GraphQL we cannot uniquely identify resources on the web but which is possible using URIs with SPARQL. In other words, GraphQL has no notion of global identifiers [15].

Talk also about differences in terms of complexity and expressivity (mathematical, theory, semantics). Read papers: Semantics and Complexity of GraphQL, Defining Schemas for Property Graphs by using the GraphQL Schema Definition Language

Chapter 3

Approaches for Querying RDF Graphs with GraphQL

In this chapter, we focus on ways of querying RDF Graphs using GraphQL. A GraphQL query has a tree-like structure. To traverse the query we can follow this tree structure, i.e., we can go up to a parent node, down to a child node or left/right to a sibling node [12]. This imitates the graph structure of by RDF data.

In section XYZ of the previous chapter, we highlighted the differences between GraphQL and SPARQL. SPARQL has advantages over GraphQL, like being more expressive owing to basic graph patterns. However it has come limitations. Considering these limitations along with the advantages that GraphQL has over SPARQL, GraphQL seems to be a prospective choice to querying RDF Graphs. For instance, the verbosity and the steep learning curve of SPARQL pose a challenge to developers in working with SPARQL. Also, the limited availability of libraries and frameworks for implementing SPARQL in applications creates a significant obstacle.

In recent years, there have been attempts at providing approaches to querying linked data represented by RDF via GraphQL. This gave rise to several commercial and open-source solutions. Most notable ones include:

- Stardog
- TopBraid EDG
- Ontotext Platform
- GraphQLSPARQL

also include this later if possible

- GraphQL-LD
- HyperGraphQL

Stardog[27] is a commercial solution that offers a graph database called "Enterprise Knowledge Graph platform"[14]. The initial versions only allowed querying their stored data using SPARQL. The support for querying using GraphQL was later added with the release of version 5.1. This solution allows users to provide a GraphQL schema. However, this is optional. When provided it can be used for translating GraphQL terms to RDF. The schema also helps in introspection purposes and for validating queries with the given typing information [**?**]. Introspection refers to querying GraphQL schema to find out the queries it supports. Moreover, schema also helps to limit the parts in the graph available to users, i.e., the user is only allowed to query some sections of the data available in the graph. If no schema is provided, then the conversion from GraphQL terms to RDF is done using the default namespace in that graph and can be overridden using the @prefix declarative inside the GraphQL queries [**?**]. However, this option requires the user to have a good knowledge about the way data is structured in the graph database. Introspection, data validation and access control is not possible when there is no schema [**?**]. In Stardog the top level node, also known as parent node, is considered to refer to a type. Any subsequent child nodes refer to the predicate links from the parent node [**?**]. This is an important distinction not necessarily shared by the other approaches.

TopBraid Enterprise Data Governance (TopBraid DG) is also a commercial solution created by TopQuadrant[28] to query RDF graphs using GraphQL. It uses GraphQL schemas that can be automatically generated based on SHACL[29] [**?**]. With the generation of schema, introspection also becomes possible. One important feature of TopBraid EDG is the functionality of query mutation. This allows the users to modify existing data in the graph.

Ontotext Platform[30] is a commercial solution offered by Ontotext. Users are provided with a GraphQL interface that they can to query the underlying graph database called "GraphDB" [14].

The above mentioned proprietary solutions are not open source and only allow querying their corresponding graph databases. This implies they cannot be used to query arbitrary knowledge graphs such as Wikidata, which is the focus of this report.

GraphQL-LD and HyperGraphQL are two open source solutions that can be used to query

[27]https://www.stardog.com/
[28]https://www.topquadrant.com/
[29]https://www.w3.org/TR/2017/REC-shacl-20170720/
[30]https://www.ontotext.com/products/ontotext-platform/

knowledge graphs that provide a SPARQL endpoint. We present in details about them in the following sections.

3.1 GraphQL-LD

GraphQL-LD originates from an ongoing research work proposed by **(author?)** to query knowledge graphs via GraphQL [15]. The main working process behind this is to extend the GraphQL queries with JSON-LD context to fetch RDF data.

JSON (Java Script Object Notation) is a widely popular data exchange format used for storing and sending information over the internet. JSON-LD (JavaScript Object Notation for Linked Data) is a syntax used to serialize Linked Data in JSON. It is also a W3C Standard. JSON-LD provides features such as identifying JSON objects by IRIs and annotating strings with their language. The official documentation for JSON-LD provided by W3C gives a detailed explanation of its features [**?**]. JSON-LD context is used to map terms into IRIs. It allows the data exchanged to be unambiguous globally in the sense that it can be meaningful to anyone receiving it on the web.

The GraphQL query selects the data (nested in fields) that we want to fetch. The JSON-LD context maps the query fields to URIs. In other words, it helps to provide a link between the items in the nodes of the queries and the actual resources that exist in the respective Linked Data source. The GraphQL queries are then converted into SPARQL queries that can be used to query any source consisting of Linked Data and a SPARQL endpoint.

Listing XYZ and Listing XYZ shows an example where a query coupled with JSON-LD context can be used to fetch RDF data from a RDF data source - http://example.org[31]. The JSON-LD context helps to identify the resources - chemicalElement, name, chemicalFormula and boilingPoint – by providing an unique IRI specific to those resources in the RDF data source.

```
{
  chemicalElement {
    name
    chemicalFormula
    boilingPoint
  }
}
```

Listing 3.1: GraphQL query

[31]This domain is for use in illustrative examples in documents

```
"@context": {
  "chemicalElement": "http://example.org/chemicalElement",
  "name": "http://example.org/name",
  "chemicalFormula": "http://example.org/chemicalFormula",
  "boilingPoint": "http://example.org/boilingPoint"
}
```

Listing 3.2: JSON-LD context

The approach taken by GraphQL-LD consists primarily of two standalone modules:

- **GraphQL to SPARQL algebra**: Parses a GraphQL query to SPARQL algebra expression
- **SPARQL results to tree**: Converts a SPARQL query result into a tree structure

3.1.1 GraphQL to SPARQL algebra

The "GraphQL to SPARQL algebra" module is used for parsing a GraphQL query into an expression in SPARQL algebra[32] with the help of a JSON-LD context. A SPARQL algebra expression is is formed from parsing the strings in SPARQL query followed by some transformations. It is basically used to provide semantics to the syntax in SPARQL query.

The algorithm for the conversion of GraphQL query to SPARQL algebra expression is based on translating the tree-like structure of GraphQL to links of triple patterns or statements in SPARQL [15]. Listing XYZ shows the SPARQL algebra obtained by parsing the GraphQL query in Listing XYZ.

```
{
  type: 'project',
  input: { type: 'bgp', patterns: [ [Quad], [Quad], [Quad], [Quad] ] },
  variables: [
    Variable { termType: 'Variable', value: 'chemicalElement_name' },
    Variable {
      termType: 'Variable',
      value: 'chemicalElement_chemicalFormula'
    },
    Variable {
      termType: 'Variable',
      value: 'chemicalElement_boilingPoint'
    }
  ]
}
```

Listing 3.3: Generated SPARQL Algebra

The Quads are listed in Listing XYZ:

[32] https://www.w3.org/TR/sparql11-query/#sparqlQuery.

```
[
  Quad {
    termType: 'Quad',
    value: '',
    subject: Variable { termType: 'Variable', value: 'df_3_0' },
    predicate: NamedNode {
      termType: 'NamedNode',
      value: 'http://example.org/chemicalElement'
    },
    object: Variable { termType: 'Variable', value: 'chemicalElement' },
    graph: DefaultGraph { termType: 'DefaultGraph', value: '' },
    type: 'pattern'
  },
  Quad {
    termType: 'Quad',
    value: '',
    subject: Variable { termType: 'Variable', value: 'chemicalElement' },
    predicate: NamedNode {
      termType: 'NamedNode',
      value: 'http://example.org/name'
    },
    object: Variable { termType: 'Variable', value: 'chemicalElement_name' },
    graph: DefaultGraph { termType: 'DefaultGraph', value: '' },
    type: 'pattern'
  },
  Quad {
    termType: 'Quad',
    value: '',
    subject: Variable { termType: 'Variable', value: 'chemicalElement' },
    predicate: NamedNode {
      termType: 'NamedNode',
      value: 'http://example.org/chemicalFormula'
    },
    object: Variable {
      termType: 'Variable',
      value: 'chemicalElement_chemicalFormula'
    },
    graph: DefaultGraph { termType: 'DefaultGraph', value: '' },
    type: 'pattern'
  },
  Quad {
    termType: 'Quad',
    value: '',
    subject: Variable { termType: 'Variable', value: 'chemicalElement' },
    predicate: NamedNode {
      termType: 'NamedNode',
      value: 'http://example.org/boilingPoint'
    },
    object: Variable {
      termType: 'Variable',
      value: 'chemicalElement_boilingPoint'
    },
    graph: DefaultGraph { termType: 'DefaultGraph', value: '' },
    type: 'pattern'
  }
]
```

Listing 3.4: The expansion of Quads

It is also possible to view the generated SPARQL queries using a CLI tool - "graphql-to-sparql". The generated SPARQL query for our example is shown in Listing XYZ.

```
SELECT ?chemicalElement_name ?chemicalElement_chemicalFormula ?
    chemicalElement_boilingPoint WHERE {
  ?df_3_0 <http://example.org/chemicalElement> ?chemicalElement.
  ?chemicalElement <http://example.org/name> ?chemicalElement_name;
    <http://example.org/chemicalFormula> ?chemicalElement_chemicalFormula;
    <http://example.org/boilingPoint> ?chemicalElement_boilingPoint.
}
```

Listing 3.5: Genertaed SPARQL query

GraphQL-LD offers many of the functionalities that are listed in the official documentation of GraphQL[33] such as fragments, directives and aliases. However, triple patterns are not sufficient to express all of these features[15]. For example, for the fragments feature in GraphQL, GraphQL-LD uses the left-join semantics. This translates to the OPTIONAL keyword in SPARQL.

Along with the "@include" and "@skip" directives[34] included in the core GraphQL specification, GraphQL-LD offers three custom directives as well to further enrich the queries - "@optional", "@single" and "plural". In GraphQL-LD all fields in the query are required to have results. In the event that any of the fields does not return a result, the entire result set will return as empty. When we are uncertain about a field returning a result we can use the "@optional" custom directive with that field. Basically, this custom directive allows the users to specify the fields that are optional. The module converts the directive to the OPTIONAL operator in SPARQL. We will discuss about the "@singular" and "plural" custom directive in the next section.

A comprehensive information on the conversion from GraphQL queries to SPARQL queries is available under the "graphql-to sparql.js" GitGub repository.[35]

3.1.2 SPARQL results to tree

The "SPARQL results to tree" module converts the SPARQL query results into a tree-based structure constituting of plain JSON objects. This is convenient since the user writes the queries in a tree-like fashion in GraphQL and expects the results to be in the same structure.

After a generated SPARQL query is sent to the Linked Data interface, such as a SPARQL endpoint, the results are returned as SPARQL JSON. This is what is meant by the SPARQL query results. This "SPARQL results to tree" module converts these results into a tree-based

[33]https://graphql.org/learn/queries/ .
[34]https://graphql.org/learn/queries/#directives.
[35]https://github.com/rubensworks/graphql-to-sparql.js

structure based by splitting the variable names based on a certain delimiter value. The default delimiter value used in GraphQL-LD is an underscore. This gives rise to paths inside the tree structure.

Listings XYZ and XYZ show an example SPARQL result and its conversion to the tree-based structure respectively.

```
{
  "results": {
    "bindings": [
      { "chemicalElement_name": { "type": "literal", "value": "Helium" }, "
    chemicalElement_chemicalFormula": { "type": "literal", "value": "He" }, "
    chemicalElement_boilingPoint": { "type": "literal", "value": "-268.9" } },
      { "chemicalElement_name": { "type": "literal", "value": "Silicon" }, "
    chemicalElement_chemicalFormula": { "type": "literal", "value": "Si" }, "
    chemicalElement_boilingPoint": { "type": "literal", "value": "4271" } }
    ]
  }
}
```

Listing 3.6: SPARQL Algebra result

```
{
"chemicalElement":[
  { "name": "Helium", "chemicalFormula": "He", "boilingPoint": "-268.9" },
  { "name": "Silicon", "chemicalFormula": "Si" , "boilingPoint": "-4271"  }
  ]
}
```

Listing 3.7: Tree-based JSON result

When writing a GraphQL query, we can decide whether the values obtained from querying should be wrapped in array or not, using the custom directives "@singluar" and "@plural" respectively on the fields. GraphQL-LD assigns all fields to be plural by default. This is a convenient feature[36] that allows the results to be compacted.

A comprehensive information on the conversion from SPARQL results to tree-strcuture is available under the "sparqljson-to-tree.js" GitGub repository.[37]

The basic overview is that GraphQL-LD takes a GraphQL query and a JSON-LD context from the user and converts the query into a SPARQL query. This generated SPARQL query is sent to a Linked Data interface such as a SPARQL endpoint for execution. We can also use this to query our own own local Linked Data files instead of a remote endpoint. Finally, the obtained query results from the endpoint are then converted into a tree-based structure corresponding to the original GraphQL query. Figure XYZ shows an overview of this approach. Here we

[36]https://github.com/rubensworks/graphql-to-sparql.js/#converting-to-tree-based-results.
[37]https://github.com/rubensworks/sparqljson-to-tree.js

show the SPARQL endpoint as the interface being queried. The modules in GraphQL-LD are implemented in TyperScript and JavaScript, and thus can be used in Javascript applications.

GraphQL-LD is predicate-oriented. This means that it focuses on querying the relationships between the subject and object [**?**]. This imposes a problem for implementing subject-based queries when we want to query for a specific class. An example for this would be querying for an item that is of type chemical element. To overcome this problem we would need a workaround. However, this makes the query complicated to write. In the next chapter where we implement querying Wikidata using GraphQL-LD, we discuss three workarounds that can used when we want to implement subject-based queries.

Since GraphQL-LD is schema-less, it not possible to perform introspection[38] - querying a GraphQL schema for information about the supported queries - since the user is not aware of the data's schema [**?**]. However, this is not necessary since all exposed Linked Data can be queried with GraphQL-LD [**?**]. When writing queries, GraphQL-LD users would need to have a good understanding of the data scheme of the queried RDF data.

3.2 HyperGraphQL

HyperGraphQL is open source GraphQL interface for querying Linked Data on the Web. It is developed and maintained by Semantics integration Ltd. The project is written in Java, with the initial release being in 2018. At the time of this writing, the latest version, 3.0.1, was released in 2021.

In addition to querying RDF data, HyperGraphQL is designed to support federated querying over multiple RDF stores via a single GraphQL query interface [**?**]. Federated queries refer to querying multiple data sources and combining the data. This enriches the results with interesting information as the data is obtained from various sources rather than just one. However, federated querying is challenging. Retrieving and combining data from different services requires deep understanding of the involved datasets, and different configuration parameters such as authentication need to be considered.

A service need to be set up that acts as an intermediary server between the client side, where the GraphQL query is written, and the RDF datastore, from where Linked Data is fetched [15] . We can create multiple instances of HyperGraphQL where each instance can query one or more RDF sources, depending on the type of Linked Data services selected. In the next section we discuss these services in detail. To set up a HyperGraphQL instance two input files need to be provided - a configuration JSON file and an annotated GraphQL schema.

[38] https://graphql.org/learn/introspection/ .

3.2.1 Configuration File

The configuration file contains the specifications of the RDF services from there data needs to be fetched. It includes the name of the instance, path of the GraphQL schema, HTTP settings of the instance and the specifications of the Linked Data services needed to fetch data. HyperGraphQL currently offers three types of Linked Data services: SPARQLEndpointService, LocalModeSPARQLService and HGraphQLService.

The SPARQLEndpointService refers to the SPARQL endpoint service where data is fetched from remote RDF sources, like Wikidata and DBpedia. LocalModeSPARQLService allows to fetch data from RDF files that exists on the local system or in a remote location. These are loaded into local memory to be used once the HyperGraphQL instance runs. These files contain data stored as RDF triples and follow the RDF serialization format of RDF/XML, Turle or N-Triples. Lastly, the HGraphQLService allows to query data from other HyperGraphQL instances running on a server.

Listing XYZ shows a configuration file for fetching data from example.org through the SPARQLEndpointService.

```
{
    "name": "example-hgql",
    "schema": "schema/schema_example.graphql",
    "server": {
        "port": 8081,
        "graphql": "/graphql",
        "graphiql": "/graphiql"
    },
    "services": [
        {
            "id": "example-sparql",
            "type": "SPARQLEndpointService",
            "url": "http://example.org/sparql/",
            "graph": "",
            "user": "",
            "password": ""
        }
    ]
}
```

Listing 3.8: An example configuration file

3.2.2 Annotated Schema

HyperGraphQL works with a GraphQL schema that needs to be defined to set up an instance. This is different from GraphQL-LD, where no schema is required. A schema gives control to the data that be queried. The user can only fetch data for the types and queries defined in the GraphQL schema [**?**].

The schema contains a designated type called "_Context". This contains the annotations that encode the mappings from every type and field in the schema to the corresponding IRI in the RDF source. The IRI should be unique and will be used to fetch data from the source.

Each type and field in the schema is annotated with GraphQL directives that tell the instance about the service from where data for the type and filed should be fetched. These are essentially pointers or service ids that correspond to the RDF services defined in the configuration file.

Only the types annotated with a service id can be queried. Non-annotated typed will not be queryable. However, all fields need be annotated with a service id. For every annotated type, two query fields are automatically exposed - TypeName_GET, parametrized with the "limit:Int" and "offset:Int" argument, and TypeName_GET_BY_ID, parameterized with "uris:[String]" argument. Two additional fields are also introduced automatically for each type in the schema - "_id:String" field which returns the IRI of the resource and "_type:String" field which returns the "rdf:type" of the parent type of the resource in the schema. Moreover, every field in the schema with the value type String is provided with a "lang:String" argument by default. This allows the user to specify the language of the fetched literal.

Listing XYZ shows a GraphQL schema that can be used to query data from example.org, using the configuration file in Listing XYZ. The id:"example-sparql" in the @service directory is the same as the id in the services block of the configuration file.

```
type __Context {
    ChemicalElement:    _@href(iri: "http://example.org/chemicalElement")
    name:               _@href(iri: "http://example.org/name")
    chemicalFormula:   _@href(iri: "http://example.org/chemicalFormula")
    boilingPoint:       _@href(iri: "http://example.org/boilingPoint")
}

type ChemicalElement @service(id:"example-sparql") {
    name: String @service(id:"example-sparql")
    chemicalFormula: String @service(id:"example-sparql")
    boilingPoint: String @service(id:"example-sparql")
}
```

Listing 3.9: An example schema

Like GraphQL-LD, HyperGraphQL also converts GraphQL queries into SPARQL. However, unlike GraphQL-LD where the conversion takes place on the client side, in HyperGraphQL the conversion takes place on the server side, thereby decreasing the computation overhead on the client side. Also, the generated SPARQL queries are not necessarily part of the user interaction, and are not displayed. We discuss in the next chapter when we implement HyperGraphQL to query Wikidata, the method we used to observe the generated SPARQL queries whenever an instance is set up.

Listing XYZ shows the GraphQL query we used in the section for GraphQL-LD. This query combined the schema in Listing XYZ generates the following SPARQL query in Listing XYZ.

```
{
  chemicalElement {
    name
    chemicalFormula
    boilingPoint
  }
}
```

Listing 3.10: An example query

```
SELECT * WHERE {
  {
      SELECT ?x_1 WHERE {
        ?x_1 http://www.w3.org/1999/02/22-rdf-syntax-ns#type http://example.org/
    chemicalElement .
    }
  }
  OPTIONAL {
    ?x_1 <http://example.org/name> ?x_1_1 .
  }
  OPTIONAL {
    ?x_1 <http://example.org/chemicalFormula> ?x_1_2 .
  }
  OPTIONAL {
    ?x_1 <http://example.org/boilingPoint> ?x_1_3 .
  }
}
```

Listing 3.11: The generated SPARQL query

We discussed in the previous section we discussed that in GraphQL-LD implementation, the result set is empty if any of the fields does not fetch a result (if the optional tag is not given to the field in the GraphQL query). With HyperGraphQL this is relaxed since the fields are translated into optional SPARQL triple patterns. We will discuss the differences between the queries generated with GraphQL-LD and HyperGraphQL in detail in chapter XYZ.

Once the instance runs on the local server, a GraphiQL[39] interface is created. This is the basically a graphical user interface where the user can write GraphQL queries and send them to the HyperGraphQL instance running on the local server. The instance queries the RDF data and returns a response in JSON-LD enhanced with JSON-LD context. Other RDF serialization formats are also supported for the returned response like json+rdf+xml and json+turtle.

[39]https://github.com/graphql/graphiql.

Chapter 4

Implementing the approaches on Wikidata

In this chapter we discuss and demonstrate the implementation of querying the knowledge graph Wikidata using both GraphQL-LD and HyperGraphQL. This is followed by the technical details and the setup of both the tools. Then, we conclude the chapter by discussing the differences between both the approaches and their limitations respectively.

4.1 GraphQL-LD on Wikidata

Wikidata offers a SPARQL endpoint https://query.wikidata.org/sparql against which requests can be sent to query its data. In GraphQL-LD, we require a GraphQL query and a JSON-LD context as input files. To demonstrate the implementation we use a practical example where we use the query and JSON-LD context shown in Listings XYZ and XYZ respectively.

Basically, we want to fetch all the chemical elements from Wikidata and some of their properties. These properties are the chemical formula, boiling point, melting point and density of the elements, along with the person who discovered or invented the elements, and that person's place of birth and country to which the place belongs. This example was used in Listing 2 where we demonstrated the querying of Wikidata using SPARQL. The only difference is that we do not query for the labels of the chemical elements.

GraphQL-LD does not support the feature of filtering the labels in a specific language. Querying for them would fetch the label for each element in all the available languages in Wikidata for that element. This would unnecessary populate the results without much benefit. Hence, we remove the label field.

```
query {
    checimcalElement @single(scope: all) {
        id
        chemicalFormula
        boilingPoint
        meltingPoint
        density
        discoverer {
            id
            placeBirth {
               id
                country {
          id
        }
            }
        }
    }
}
```

Listing 4.1: Query

```
{
  "@context": {
    "wd": "http://www.wikidata.org/entity/",
    "wdt":"http://www.wikidata.org/prop/direct/",
    "instance": "wdt:P31",
    "chemicalFormula": "wdt:P274",
    "boilingPoint": "wdt:P2102",
    "meltingPoint": "wdt:P2101",
    "density": "wdt:P2054",
    "discoverer": "wdt:P61",
    "placeBirth": "wdt:P19",
    "country": "wdt:P17"
  }
}
```

Listing 4.2: JSON-LD Context

The generated SPARQL query is shown in Listing XYZ.

```
SELECT ?checimcalElement_id ?checimcalElement_chemicalFormula ?
    checimcalElement_boilingPoint ?checimcalElement_meltingPoint ?
    checimcalElement_density ?checimcalElement_discoverer_id
?checimcalElement_discoverer_placeBirth_id ?
    checimcalElement_discoverer_placeBirth_country WHERE {
  ?df_3_0 undefined:checimcalElement ?checimcalElement_id.
  ?checimcalElement_id <http://www.wikidata.org/prop/direct/P274> ?
    checimcalElement_chemicalFormula;
    <http://www.wikidata.org/prop/direct/P2102> ?checimcalElement_boilingPoint;
    <http://www.wikidata.org/prop/direct/P2101> ?checimcalElement_meltingPoint;
    <http://www.wikidata.org/prop/direct/P2054> ?checimcalElement_density;
    <http://www.wikidata.org/prop/direct/P61> ?checimcalElement_discoverer_id.
  ?checimcalElement_discoverer_id <http://www.wikidata.org/prop/direct/P19> ?
    checimcalElement_discoverer_placeBirth_id.
  ?checimcalElement_discoverer_placeBirth_id <http://www.wikidata.org/prop/direct/P17> ?
    checimcalElement_discoverer_placeBirth_country.
}
```

Listing 4.3: Generated SPARQL Query

The above SPARQL query does not produce any results in Wikidata. As we mentioned in chapter XYZ, GraphQL-LD is predicate-oriented. It queries the relationships between nodes. This creates a problem when we want to say that some subject is an instance or type of some object. GraphQL-LD does not recognize the root node to be an instance or type of a node. Instead it treats the root node to be a property. Hence, some workaround needs to be done to achieve the desired results.

We propose three solutions that can be potentially use as a workaround.

4.1.1 Inline-ID based Solution 1

This solution is inspired by the "Setting an inline id"[40] section provided in the documentation of GraphQL-LD. Although it is intended to be used for defining or looking up the id of entities as per the documentation, we realized this option come help overcome the issue we were having when defining an item to be an instance of another item. Listings XYZ and XYZ show the GraphQL query and the corresponding generated SPARQL query generated using the JSON-LD context in Listing XYZ respectively.

To query against Wikidata, we can provide the available SPARQL endpoint – https://query.wikidata.org/sparql. This generates the results in JSON shown in Listing XYZ. We show only the first three results. The LIMIT solution modifier could have been implemented for this by using "first: 3" in our GraphQL queries. However, GraphQL-LD would have generated a nested SPARQL query, and we decided that the comparison between the proposed solutions would be more prominent and easier to understand using a simpler SPARQL

[40]https://github.com/rubensworks/graphql-to-sparql.js#setting-an-inline-id.

query.

```
query {
    instance(_: chemicalElement) @single(scope: all)
        id
        chemicalFormula
        boilingPoint
        meltingPoint
        density
        discoverer {
            id
            placeBirth {
                id
                country
            }
        }
}
```

Listing 4.4: Query

```
SELECT ?id ?chemicalFormula ?boilingPoint ?meltingPoint ?density ?discoverer_id ?
    discoverer_placeBirth_id ?discoverer_placeBirth_country WHERE {
  ?id <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/Q11344>;
    <http://www.wikidata.org/prop/direct/P274> ?chemicalFormula;
    <http://www.wikidata.org/prop/direct/P2102> ?boilingPoint;
    <http://www.wikidata.org/prop/direct/P2101> ?meltingPoint;
    <http://www.wikidata.org/prop/direct/P2054> ?density;
    <http://www.wikidata.org/prop/direct/P61> ?discoverer_id.
  ?discoverer_id <http://www.wikidata.org/prop/direct/P19> ?discoverer_placeBirth_id.
  ?discoverer_placeBirth_id <http://www.wikidata.org/prop/direct/P17> ?
    discoverer_placeBirth_country.
}
```

Listing 4.5: Generated SPARQL Query

```
[
  {
    "id": "http://www.wikidata.org/entity/Q560",
    "boilingPoint": -268.9,
    "density": 0.1785,
    "meltingPoint": -272.05,
    "discoverer": {
      "id": "http://www.wikidata.org/entity/Q298581",
      "placeBirth": {
        "id": "http://www.wikidata.org/entity/Q90",
        "country": "http://www.wikidata.org/entity/Q142"
      }
    },
    "chemicalFormula": "He"
  },
  {
    "id": "http://www.wikidata.org/entity/Q1119",
    "boilingPoint": 8316,
    "density": 13,
    "meltingPoint": 4041,
    "discoverer": {
      "id": "http://www.wikidata.org/entity/Q775969",
      "placeBirth": {
        "id": "http://www.wikidata.org/entity/Q727",
        "country": "http://www.wikidata.org/entity/Q55"
      }
    },
    "chemicalFormula": "Hf"
  },
  {
    "id": "http://www.wikidata.org/entity/Q1094",
    "boilingPoint": 3767,
    "density": 7.31,
    "meltingPoint": 314,
    "discoverer": {
      "id": "http://www.wikidata.org/entity/Q77308",
      "placeBirth": {
        "id": "http://www.wikidata.org/entity/Q1731",
        "country": "http://www.wikidata.org/entity/Q183"
      }
    },
    "chemicalFormula": "In"
  },
  ...
]
```


Listing 4.6: Output

4.1.2 ID based Solution 2

This solution is proposed after doing some experiments with the source code. The documentation from GraphQL-LD does not provide any reference to this but we consider this to be a potential solution. Initially, when no id was provided as a field in the GraphQL query, the

fetched result only contained ids of the queried item, and ignored the rest of the properties. We had to update the source code to overcome this issue.

Listings XYZ, XYZ and XYZ show the GraphQL query, generated SPARQL query and the corresponding first three JSON results after querying Wikidata. The only issue is that the generated SPARQL query includes the id in the SELECT clause even if no id is provided in the SPARQL query, although this does not affect the results and we get them as expected. We can try to correct this issue in future works by updating the source code further.

```
query {
    id (instance: chemicalElement) @single(scope: all) {
        id
        chemicalFormula
        boilingPoint
        meltingPoint
        density
        discoverer {
            placeBirth {
                id
                country
            }
        }
    }
}
```

Listing 4.7: Query

```
SELECT ?id ?id_id ?id_chemicalFormula ?id_boilingPoint ?id_meltingPoint ?id_density ?
    id_discoverer_placeBirth_id ?id_discoverer_placeBirth_country WHERE {
  ?id_id <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/
    Q11344>;
    <http://www.wikidata.org/prop/direct/P274> ?id_chemicalFormula;
    <http://www.wikidata.org/prop/direct/P2102> ?id_boilingPoint;
    <http://www.wikidata.org/prop/direct/P2101> ?id_meltingPoint;
    <http://www.wikidata.org/prop/direct/P2054> ?id_density;
    <http://www.wikidata.org/prop/direct/P61> ?id_discoverer.
  ?id_discoverer <http://www.wikidata.org/prop/direct/P19> ?id_discoverer_placeBirth_id.
  ?id_discoverer_placeBirth_id <http://www.wikidata.org/prop/direct/P17> ?
    id_discoverer_placeBirth_country.
}
```

Listing 4.8: Generated SPARQL Query

```
[
  {
    "id": {
      "id": "http://www.wikidata.org/entity/Q560",
      "boilingPoint": -268.9,
      "density": 0.1785,
      "meltingPoint": -272.05,
      "chemicalFormula": "He",
      "discoverer": {
        "placeBirth": {
          "id": "http://www.wikidata.org/entity/Q90",
          "country": "http://www.wikidata.org/entity/Q142"
        }
      }
    }
  },
  {
    "id": {
      "id": "http://www.wikidata.org/entity/Q1119",
      "boilingPoint": 8316,
      "density": 13,
      "meltingPoint": 4041,
      "chemicalFormula": "Hf",
      "discoverer": {
        "placeBirth": {
          "id": "http://www.wikidata.org/entity/Q727",
          "country": "http://www.wikidata.org/entity/Q55"
        }
      }
    }
  },
  {
    "id": {
      "id": "http://www.wikidata.org/entity/Q1094",
      "boilingPoint": 3767,
      "density": 7.31,
      "meltingPoint": 314,
      "chemicalFormula": "In",
      "discoverer": {
        "placeBirth": {
          "id": "http://www.wikidata.org/entity/Q1731",
          "country": "http://www.wikidata.org/entity/Q183"
        }
      }
    }
  },
  ...
]
```


Listing 4.9: Output

4.1.3 Fragment based Solution 3

This last solution is inspired by the inline fragment[41] usage of GraphQL-LD. Initially, GraphQL-LD translated the "id ... on chemicalElement" part as "?id rdf:type <IRI of chemicalElement>".[42] This triple pattern does not correspond to the data model in Wikidata as it uses the "instance of" property instead of "rdf:type" when describing that an item is a type of a class. To overcome this issue we had to modify the source code so that it used "http://www.wikidata.org/prop/direct/P31" instead of "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>" when translating GraphQL queries to SPARQL queries when fragments are used. Unlike the other two proposed solutions where the "instance" property must be used in the queries, this solution does not require it. Consequently, no entries for the mapping of "instance" of to its IRI, http://www.wikidata.org/entity/Q11344), is needed in the JSON-LD context.

Listings XYZ, XYZ and XYZ show the GraphQL query, generated SPARQL query and the corresponding first three JSON results fetched from Wikidata. This solution treats the fragments as OPTIONAL patterns in SPARQL. Since it is based on fragmentation, the id is always fetched.

```
query {
    id
    ... on chemicalElement @single(scope: all) {
        id
        chemicalFormula
        boilingPoint
        meltingPoint
        density
        discoverer {
            placeBirth {
                id
                country
            }
        }
    }
}
```

Listing 4.10: Query

[41] https://github.com/rubensworks/graphql-to-sparql.js#inline-fragments.

[42] For our query this would be "?id <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.wikidata.org/entity/Q11344>"

```
SELECT ?id ?id_id ?chemicalFormula ?boilingPoint ?meltingPoint ?density ?
    discoverer_placeBirth_id ?discoverer_placeBirth_country WHERE {
  OPTIONAL {
    ?id <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/Q11344
    >;
      <http://www.wikidata.org/prop/direct/P274> ?chemicalFormula;
      <http://www.wikidata.org/prop/direct/P2102> ?boilingPoint;
      <http://www.wikidata.org/prop/direct/P2101> ?meltingPoint;
      <http://www.wikidata.org/prop/direct/P2054> ?density;
      <http://www.wikidata.org/prop/direct/P61> ?discoverer.
    ?discoverer <http://www.wikidata.org/prop/direct/P19> ?discoverer_placeBirth_id.
    ?discoverer_placeBirth_id <http://www.wikidata.org/prop/direct/P17> ?
    discoverer_placeBirth_country.
  }
}
```

Listing 4.11: Generated SPARQL Query

```
[
  {
    "id": "http://www.wikidata.org/entity/Q560",
    "boilingPoint": -268.9,
    "density": 0.1785,
    "meltingPoint": -272.05,
    "chemicalFormula": "He",
    "discoverer": {
      "placeBirth": {
        "id": "http://www.wikidata.org/entity/Q90",
        "country": "http://www.wikidata.org/entity/Q142"
      }
    }
  },
  {
    "id": "http://www.wikidata.org/entity/Q1119",
    "boilingPoint": 8316,
    "density": 13,
    "meltingPoint": 4041,
    "chemicalFormula": "Hf",
    "discoverer": {
      "placeBirth": {
        "id": "http://www.wikidata.org/entity/Q727",
        "country": "http://www.wikidata.org/entity/Q55"
      }
    }
  },
  {
    "id": "http://www.wikidata.org/entity/Q1094",
    "boilingPoint": 3767,
    "density": 7.31,
    "meltingPoint": 314,
    "chemicalFormula": "In",
    "discoverer": {
      "placeBirth": {
        "id": "http://www.wikidata.org/entity/Q1731",
        "country": "http://www.wikidata.org/entity/Q183"
      }
    }
  },
  ...
]
```


Listing 4.12: Output

The above three proposed solution are prospective workarounds for describing the "instance of" relationship between two items in Wikidata. The first two solutions, Inline-ID based and ID based, are more feasible than the Fragment based solution. The way of describing the “instance of” relationship using inline fragments in the third solution is not intuitive, as they are primarily used to query a field that returns an interface or a union type.[43] Between the first two solutions, the second one namely the ID based solution is more intuitive in writing GraphQL queries that

[43]https://graphql.org/learn/queries/#inline-fragments.

the first solution.

4.1.4 Default JSON-LD context

Working with GraphQL-LD requires creating a JSON-LD context that requires some significant effort to create. The IRIs of the items used in the GraphQL query need to be looked up and then inserted into the context. To overcome this constraint we created a default JSON-LD context. This contains a list of all items and properties along with their IRIs available in Wikidata as of 12.10.2022. Also, the source code of GraphQL-LD was modified such that the tool takes the default context as input when no context is provided by the user. This default context was stored in a local directory.

For the creation of such a default context, we needed to extract all the items and properties in Wikidata along with their labels to identify them. For the items, we downloaded the file consisting of all truthy statements[44] from the RDF dumps provided by Wikidata.[45] We extracted all the items with their labels from this file and removed duplicates. Since many items can have the same labels, there would be ambiguity when the query involved one of these items. Hence, we remove all instances of items where the duplicates occurred. For the properties, we used a SPARQL query to fetch all the properties in Wikidata along with their labels. Finally, a JSON-LD context was created consisting of all the items, properties and their labels. In total there are 84,535,915 items and 10,317 properties in our default context. For the entire process, we used Python and shell scripts.

4.2 HyperGraphQL on Wikidata

To query RDF data using HyperGraphQL we need to set up a HyperGraphQL instance. This requires a configuration file and an annotated schema. We want to query Wikidata for the same data as we did in Listing XYZ when using GraphQL-LD, except now we can query for labels too as HyperGraphQL support it. Listings XYZ, XYZ and XYZ show the GraphQL query, configuration file and schema. We use the same SPARQL endpoint for Wikidata as we did in GraphQl-LD.

[44] https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format#Truthy_statements.
[45] https://dumps.wikimedia.org/wikidatawiki/entities/ .

```
{
  ChemicalElement_GET {
    _id
    label(lang: "en")
    chemicalFormula
    boilingPoint
    meltingPoint
    density
    discoverer {
      _id
      label(lang: "en")
      placeBirth {
        _id
        label(lang: "en")
        country {
          _id
          label(lang: "en")
        }
      }
    }
  }
}
```

Listing 4.13: Query

```
{
    "name": "wikidata-hgql",
    "schema": "schema/schema_wikidata.graphql",
    "server": {
        "port": 8081,
        "graphql": "/graphql",
        "graphiql": "/graphiql"
    },
    "services": [
        {
            "id": "wikidata-sparql",
            "type": "SPARQLEndpointService",
            "url": "https://query.wikidata.org/sparql",
            "graph": "",
            "user": "",
            "password": ""
        }
    ]
}
```

Listing 4.14: Configuration

```
type __Context {
    ChemicalElement:   _@href(iri: "http://www.wikidata.org/entity/Q11344")
    label:  _@href(iri: "http://www.w3.org/2000/01/rdf-schema#label")
    chemicalFormula:  _@href(iri: "http://www.wikidata.org/prop/direct/P274")
    boilingPoint:    _@href(iri: "http://www.wikidata.org/prop/direct/P2102")
    meltingPoint:  _@href(iri: "http://www.wikidata.org/prop/direct/P2101")
    density:  _@href(iri: "http://www.wikidata.org/prop/direct/P2054")
    discoverer:    _@href(iri: "http://www.wikidata.org/prop/direct/P61")
    Human:  _@href(iri: "http://www.wikidata.org/entity/Q5")
    placeBirth:    _@href(iri: "http://www.wikidata.org/prop/direct/P19")
    City: _@href(iri: "http://www.wikidata.org/entity/Q515")
    country:    _@href(iri: "http://www.wikidata.org/prop/direct/P17")
    SovereignState:    _@href(iri: "http://www.wikidata.org/entity/Q3624078")
}

type ChemicalElement @service(id:"wikidata-sparql") {
    chemicalFormula: String! @service(id:"wikidata-sparql")
    label: [String] @service(id:"wikidata-sparql")
    boilingPoint: String @service(id:"wikidata-sparql")
    meltingPoint: String @service(id:"wikidata-sparql")
    density: String @service(id:"wikidata-sparql")
    discoverer: [Human] @service(id:"wikidata-sparql")
}

type Human @service(id:"wikidata-sparql"){
    placeBirth: [City] @service(id:"wikidata-sparql")
    label: [String] @service(id:"wikidata-sparql")
}

type City @service(id:"wikidata-sparql"){
    country: [SovereignState] @service(id:"wikidata-sparql")
    label: String @service(id:"wikidata-sparql")
}

type SovereignState @service(id:"wikidata-sparql"){
    label: [String] @service(id:"wikidata-sparql")
}
```

Listing 4.15: Schema

Once an HyperGraphQL instance is set up, we can write the GraphQL query in the provided graphiql interface that can simply be run in a browser. After the query is executed, the instance fetches the results from Wikidata and displays them in the interface.

HyperGraphQL translates the root node to be an rdf:type of some subject. Since this does not comply with the Wikidata data model, we had a similar issue like the one we discussed in the Fragment based Solution 3 of the previous section regarding inline fragments in GraphQL-LD. To overcome this issue we had to update the source code so that the generated SAPRQL queries use instance of property instead of rdf:type.

The generated SPARQL queries are not viewable by default. We had to modify the logging option in the source code to log SPARQL queries on the command line. Listings XYZ and

XYZ show the generated SPARQL query and the results fetched. As before we show only the first 3 results.

In HyperGraphQL, the fields are converted into optional SPARQL triple patterns. Hence, it might be possible to have an empty array in the results.

```
SELECT * WHERE {
  {
    SELECT ?x_1 WHERE {
      ?x_1 <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/
    Q11344> .
    }
  }
  OPTIONAL {
    ?x_1 <http://www.w3.org/2000/01/rdf-schema#label> ?x_1_1 .
    FILTER (lang(?x_1_1) = "en") .
  }
  OPTIONAL {
    ?x_1 <http://www.wikidata.org/prop/direct/P2101> ?x_1_2 .
  }
  OPTIONAL {
    ?x_1 <http://www.wikidata.org/prop/direct/P2054> ?x_1_3 .
  }
  OPTIONAL {
    ?x_1 <http://www.wikidata.org/prop/direct/P2102> ?x_1_4 .
  }
  OPTIONAL {
    ?x_1 <http://www.wikidata.org/prop/direct/P61> ?x_1_5 .
    ?x_1_5 <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/Q5>
     .
    OPTIONAL {
      ?x_1_5 <http://www.w3.org/2000/01/rdf-schema#label> ?x_1_5_1 .
      FILTER (lang(?x_1_5_1) = "en") .
    }
    OPTIONAL {
      ?x_1_5 <http://www.wikidata.org/prop/direct/P19> ?x_1_5_2 .
      ?x_1_5_2 <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity
    /Q515> .
      OPTIONAL {
        ?x_1_5_2 <http://www.w3.org/2000/01/rdf-schema#label> ?x_1_5_2_1 .
        FILTER (lang(?x_1_5_2_1) = "en") .
      }
      OPTIONAL {
        ?x_1_5_2 <http://www.wikidata.org/prop/direct/P17> ?x_1_5_2_2 .
        ?x_1_5_2_2 <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/
    entity/Q3624078> .
        OPTIONAL {
          ?x_1_5_2_2 <http://www.w3.org/2000/01/rdf-schema#label> ?x_1_5_2_2_1 .
          FILTER (lang(?x_1_5_2_2_1) = "en") .
        }
      }
    }
  }
  OPTIONAL {
    ?x_1 <http://www.wikidata.org/prop/direct/P274> ?x_1_6 .
  }
}
```

Listing 4.16: Generated SPARQL Query

```
{
  "extensions": {},
  "data": {
    "@context": {
      "placeBirth": "http://www.wikidata.org/prop/direct/P19",
      "country": "http://www.wikidata.org/prop/direct/P17",
      "density": "http://www.wikidata.org/prop/direct/P2054",
      "chemicalFormula": "http://www.wikidata.org/prop/direct/P274",
      "_type": "@type",
      "_id": "@id",
      "label": "http://www.w3.org/2000/01/rdf-schema#label",
      "discoverer": "http://www.wikidata.org/prop/direct/P61",
      "meltingPoint": "http://www.wikidata.org/prop/direct/P2101",
      "ChemicalElement_GET": "http://hypergraphql.org/query/ChemicalElement_GET",
      "boilingPoint": "http://www.wikidata.org/prop/direct/P2102"
    },
    "ChemicalElement_GET": [
      {
        "_id": "http://www.wikidata.org/entity/Q876",
        "label": [
          "selenium"
        ],
        "chemicalFormula": "[Se]",
        "boilingPoint": "[1265]",
        "meltingPoint": "[392]",
        "density": "[4.28]",
        "discoverer": [
          {
            "_id": "http://www.wikidata.org/entity/Q353490",
            "label": [
              "Johan Gottlieb Gahn"
            ],
            "placeBirth": []
          },
          {
            "_id": "http://www.wikidata.org/entity/Q151911",
            "label": [
              "J\"{o}ns Jacob Berzelius"
            ],
            "placeBirth": []
          }
        ]
      },
      {
        "_id": "http://www.wikidata.org/entity/Q54377",
        "label": [
          "unbihexium"
        ],
        "chemicalFormula": "[]",
        "boilingPoint": "[]",
        "meltingPoint": "[]",
        "density": "[]",
        "discoverer": []
      },
      {
        "_id": "http://www.wikidata.org/entity/Q743",
        "label": [
          "tungsten"
        ],
        "chemicalFormula": "[W]",
        "boilingPoint": "[5930, 10701]",
        "meltingPoint": "[6170, 3410]",
        "density": "[19.3]",
        "discoverer": [
          {
```

4.3 Differences and Limitations

Query, Functionality, Performance, Limitations

4.4 Technicalities and setup

There were some challenges encountered when trying to test GraphQL-LD and HypergraphQL, and using them to query Wikidata.

With GraphQL-LD, there were issues running the examples provided in the official documentation. These were related with compatibility when invoking the programmatic API written in ES6 on modules written in CommonJS. To solve this issue we had to update the API codes and the root level node package. Since GraphQL-LD is predicate based, it was challenging to find a workaround to query Wikidata. There were no examples on the documentation for subject-based querying. A significant amount of time and research was invested in finding out the three proposed solutions. The process of creating a default context required a lot of time and resource consumption. The original dump file of truthy statements was around 60GB, and hence required a large bandwidth and memory consumption for downloading and parsing the file.

An important part of our work is to analyze the generated SPARQL queries. When working with HyperGraphQL, the viewing of the generated SPARQL queries is not available by default. We had to study the source code in detail in order to log them on the command line.

We discussed in the previous chapter about the data model in Wikidata where the RDF property name "rdf:type" is replaced by Wikidata property "instance of". This property is used when the subject is an instance of a class. In GraphQL-LD, the fragments in GraphQL queries apply on types. When translating to SPARQL queries, the predicate rdf:type is used to connect the subject to some type of object. We had to analyze the "GraphQL to SPARQL algebra" module in order to update the source code so that GraphQL-LD used "instance of" instead of "rdf:type". Similarly, HyperGraphQL also used rdf:type as the property when translating the root node in GraphQL queries to SPARQL. A similar effort was needed to update the source code such that "instance of" was used instead of "rdf:type" so that data from Wikidata could be queried.

We have created a public repository[46] where all the changes made to the source code of both the approaches, GraphQL-LD and HyperGraphQL, are retained. This repository can be used to test the querying of Wikidata via both of the approaches. Along with containing the installation guide to use the repository, it contains examples to query Wikidata including the ones used in

[46]https://github.com/AnasShahab/querying-wikidata-with-graphQL.

this report.

Chapter 5

Evaluation of the generated SPARQL queries

I think the title is not very clear on a high level

We now evaluate the SPARQL queries generated by each of the two approaches - GraphQL-LD and HyperGraphQL. This is done by using ten SPARQL queries taken from the official Wikidata documentation[47] and analysing whether these can also be generated using the two approaches. Any significant differences between the SPARQL queries are also compared. The queries used for evaluation are selected based on the different functionalities of SPARQL that can be used for fetching data from Wikidata.

In Wikidata we can use the specialized service with the URI <http://wikiba.se/ontology#label> to fetch labels in specific languages. We cannot use this service in any of the approaches, and rdf:label[48] is used instead. GraphQL-LD cannot filter to fetch labels in some particular languages. It retrieves labels in all the available languages for the Wikidata resource. On the other hand, HyperGraphQL can fetch labels but do filtering to fetch labels in any one language.

HyperGraphQL translates fields into

typo

OPTIOANAL SPARQL triple patterns. So we would get more results than expected when writing GraphQL queries. Also, the use of a schema in HyperGraphQL restricts the object node item in a triple pattern to be an instance of some class. This restriction results in fewer exploration of data, as the object node could have been an instance of other classes too. This is

[47] https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples.
[48] http://www.w3.org/2000/01/rdf-schema#label.

evident in Query 8

Query 1 fetches at all items whose value of instance of is house cat.

a little hard to read; highlight “instance of” somehow (applies throughout the whole report)

```
SELECT ?item ?itemLabel
WHERE
{
    ?item wdt:P31 wd:Q146;.
    SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
```

Listing 5.1: Query 1

```
SELECT ?id ?id_id ?id_label WHERE {
  ?id_id <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/Q146
    >;
    <http://www.w3.org/2000/01/rdf-schema#label> ?id_label.
}
```

Listing 5.2: Query 1 - GraphQL-LD

```
SELECT * WHERE {
  {
    SELECT ?x_1 WHERE {
      ?x_1 <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/
    Q146> .
    }
  }
  OPTIONAL {
    ?x_1 <http://www.w3.org/2000/01/rdf-schema#label> ?x_1_1 .
    FILTER (lang(?x_1_1) = "en") .
  }
}
```

Listing 5.3: Query 1 - HypergraphQL

Query 2 counts the total number of humans in Wikidata using COUNT function.

```
SELECT (COUNT(*) AS ?count)
WHERE {
  ?item wdt:P31 wd:Q5 .
}
```

Listing 5.4: Query 2

```
SELECT ?id ?id_totalCount WHERE {
  SELECT (COUNT(?id) AS ?id_totalCount) WHERE {
    ?id <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/Q5>.
  }
}
```

Listing 5.5: Query 2 - GraphQL-LD

Query 2 cannot be impplemented with HyperGraphQL. HyperGraphQL does not support the use of functions. As a result, the query using the Count function cannot be implemented in HyperGraphQL.

Query 3 fetches Humans that do not have children using the "no value" handling, and considers only truthy values.

```
SELECT ?human ?humanLabel
WHERE
{
  ?human wdt:P31 wd:Q5
  ?human rdf:type wdno:P40 .
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en" }
}
```

Listing 5.6: Query 3

```
SELECT ?id ?id_id ?id_label WHERE {
  ?id_id <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/Q5>;
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.wikidata.org/prop/
    novalue/P40>;
    <http://www.w3.org/2000/01/rdf-schema#label> ?id_label.
}
```

Listing 5.7: Query 3 - GraphQL-LD

This query cannot be written in HyperGraphQL. The "no value" is represented as a class of an entity, statement or reference, and is defined to be of type owl:Class[49] Since we have modified the HyperGraphQL source code to use instance of property instead of rdf type in the schema, we cannot define "no value" correctly.

Query 4 in similar to Query 3 but it also considers non-truthy "no-value" statement to fetch Humans without children in Wikidata.

[49]http://www.w3.org/2002/07/owl#Class.

```
SELECT ?human ?humanLabel
WHERE
{
  ?human wdt:P31 wd:Q5
  ?human p:P40 ?childStatement .
  ?childStatement rdf:type wdno:P40 .
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en" . }
}
```

Listing 5.8: Query 4

```
SELECT ?id ?id_id ?id_childStatement ?id_label WHERE {
  ?id_id <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/Q5>;
    <http://www.wikidata.org/prop/P40> ?id_childStatement.
  ?id_childStatement <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://www.
    wikidata.org/prop/novalue/P40>.
  ?id_id <http://www.w3.org/2000/01/rdf-schema#label> ?id_label.
}
```

Listing 5.9: Query 4 - GraphQL-LD

Query 4 cannot be written in HyperGraphQL due to similar reasons as shown in Query 3.

Query 5 fetches the first 10 items in Wikidata that have the Wikimedia database name property.

```
SELECT ?item ?itemLabel ?value
WHERE
{
  ?item wdt:P1800 ?value
   SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
LIMIT 10
```

Listing 5.10: Query 5

```
SELECT ?id ?id_wikimediaDatabase WHERE {
  SELECT ?id ?id_wikimediaDatabase WHERE { ?id <http://www.wikidata.org/prop/direct/
    P1800> ?id_wikimediaDatabase. }
  LIMIT 10
}
```

Listing 5.11: Query 5 - GraphQL-LD

```
SELECT * WHERE {
  {
    SELECT ?x_1 WHERE
    {
      ?x_1 <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/
    Q33120876> .
    } LIMIT 10
  }
  OPTIONAL {
     ?x_1 <http://www.w3.org/2000/01/rdf-schema#label> ?x_1_1 .
     FILTER (lang(?x_1_1) = "en") .
    }
    OPTIONAL {
      ?x_1 <http://www.wikidata.org/prop/direct/P1800> ?x_1_2 .
    }
}
```

Listing 5.12: Query 5 - HypergraphQL

Query 6 fetches mayors that are either a dog, a cat or a chicken by using the VALUES clause.

```
SELECT ?image ?speciesLabel ?mayorLabel ?placeLabel WHERE {
  VALUES ?species {wd:Q144 wd:Q146 wd:Q780}
  ?mayor wdt:P31 ?species .
  ?mayor p:P39 ?node .
  ?node ps:P39 wd:Q30185 .
  ?node pq:P642 ?place .
  OPTIONAL {?mayor wdt:P18 ?image}
  SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
```

Listing 5.13: Query 6

Query 6 cannot be written in either GraphQL-LD or HyperGraphQL. GraphQL-LD does not support the use of VALUES clause. HyperGraphQL only supports the binding of subject variable in the root level to values. We have demonstrated this in Query 8.

Query 7 retrieves all the subclass of literary work in Wikidata.

```
SELECT ?s ?desc
WHERE
{
  ?s wdt:P279 wd:Q7725634 .
  OPTIONAL {
     ?s rdfs:label ?desc FILTER (lang(?desc) = "en").
   }
 }
```

Listing 5.14: Query 7

```
SELECT ?id ?label WHERE {
  ?id <http://www.wikidata.org/prop/direct/P279> <http://www.wikidata.org/entity/
    Q7725634>.
  OPTIONAL { ?id <http://www.w3.org/2000/01/rdf-schema#label> ?label. }
}
```

Listing 5.15: Query 7 - GraphQL-LD

Query 7 cannot be written in HyperGraphQL since values cannot be set to object nodes.

Query 8 fetches the destinations that can be reached from Antwerp International airport using the VALUES clause.

```
SELECT ?connectsairport ?connectsairportLabel ?place_served ?place_servedLabel ?coor
WHERE
{
  VALUES ?airport { wd:Q17480 }
?airport wdt:P81 ?connectsairport ;
           wdt:P625 ?base_airport_coor .
  ?connectsairport wdt:P931 ?place_served ;
                   wdt:P625 ?coor .

  SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
}
```

Listing 5.16: Query 8

Query 8 cannot be written in GraphQL-LD due to the same reasons provided in Query 6.

```
SELECT * WHERE {
  VALUES ?x_1 { <http://www.wikidata.org/entity/Q17480> }
  ?x_1 <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/
    Q1248784> .
  OPTIONAL {
    ?x_1 <http://www.wikidata.org/prop/direct/P81> ?x_1_1 .
    ?x_1_1 <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity/
    Q1248784> .
    OPTIONAL {
      ?x_1_1 <http://www.w3.org/2000/01/rdf-schema#label> ?x_1_1_1 .
      FILTER (lang(?x_1_1_1) = "en") .
    }
    OPTIONAL {
      ?x_1_1 <http://www.wikidata.org/prop/direct/P931> ?x_1_1_2 .
      ?x_1_1_2 <http://www.wikidata.org/prop/direct/P31> <http://www.wikidata.org/entity
    /Q515> .
      OPTIONAL {
        ?x_1_1_2 <http://www.w3.org/2000/01/rdf-schema#label> ?x_1_1_2_1 .
        FILTER (lang(?x_1_1_2_1) = "en") .
      }
      OPTIONAL {
        ?x_1_1_2 <http://www.wikidata.org/prop/direct/P625> ?x_1_1_2_2 .
      }
    }
  }
  OPTIONAL {
    ?x_1 <http://www.w3.org/2000/01/rdf-schema#label> ?x_1_2 .
    FILTER (lang(?x_1_2) = "en") .
  }
}
```

Listing 5.17: Query 8 - HypergraphQL

Query 9 fetches the locations of the works created by Pablo Picasso.

```
SELECT ?label ?coord ?subj
WHERE
{
   ?subj wdt:P170 wd:Q5593 .
  OPTIONAL {?subj wdt:P276 ?loc .
    ?loc wdt:P625 ?coord } .
   ?subj rdfs:label ?label FILTER (lang(?label) = "en")
}
```

Listing 5.18: Query 9

```
SELECT ?id ?id_location_coordinateLocation ?id_label WHERE {
  ?id <http://www.wikidata.org/prop/direct/P170> <http://www.wikidata.org/entity/Q5593>;
    <http://www.w3.org/2000/01/rdf-schema#label> ?id_label.
  OPTIONAL {
    ?id <http://www.wikidata.org/prop/direct/P276> ?id_location.
    ?id_location <http://www.wikidata.org/prop/direct/P625> ?
    id_location_coordinateLocation.
  }
}
```

Listing 5.19: Query 9 - GraphQL-LD

Query 9 cannot be written in HyperGraphQL owing to similar reasons mentioned in Query 7.

Query 10 uses VALUES to extract scientific articles of Lydia Pintscher.

```
SELECT ?entity ?entityLabel ?authorLabel WHERE {
  VALUES ?author {wd:Q18016466} #initialize "?author with the Wikidata item "Lydia
    Pintscher"
  ?entity wdt:P31 wd:Q13442814. #filter by scientific articles
  ?entity wdt:P50 ?author.
 SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en". }
}
```

Listing 5.20: Query 10

Query 10 cannot be written in either GraphQL-LD or HyperGraphQL for similar reasons as mentioned in Query 6.

Is an ending remark in evaluation needed

some kind of verdict would be necessary yes; the evaluation (and the report) should have a goal (a verdict for the goal of the report should then be given in the conclusion)

Need to include GraphQL queries too in evaluation?

yes, also give the graphql queries (but maybe you do not have to give all examples in the report if they are very similar; you should still point to the supplementary material / the repo where you have all of them)

interesting findings!

Chapter 6

Conclusion and future work

Bibliography

[1] Hogan, A., Blomqvist, E., Cochez, M., D'amato, C., Melo, G.D., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., Ngomo, A.C.N., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., Zimmermann, A.: Knowledge Graphs. ACM Comput. Surv. **54**(4) (jul 2021). https://doi.org/10.1145/3447772, https://doi.org/10.1145/3447772

[2] Ji, S., Pan, S., Cambria, E., Marttinen, P., Yu, P.S.: A Survey on Knowledge Graphs: Representation, Acquisition, and Applications. IEEE Transactions on Neural Networks and Learning Systems **33**(2), 494–514 (2022). https://doi.org/10.1109/TNNLS.2021.3070843

[3] Bordes, A., Weston, J., Collobert, R., Bengio, Y.: Learning structured embeddings of knowledge bases. In: Twenty-fifth AAAI conference on artificial intelligence (2011)

[4] Ruth, L., Wood, D., Zaidman, M., Hausenblas, M.: Linked Data: Structured data on the Web. Manning (2013), https://books.google.de/books?id=hTozEAAAQBAJ

[5] R. Cyganiak, D.W., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. Available at https://www.w3.org/TR/rdf11-concepts/ 2022-11-28 (2014)

[6] Bonduel, Mathias and Wagner, Anna and Pauwels, Pieter and Vergauwen, Maarten and Klein, Ralf: Including widespread geometry formats in semantic graphs using RDF literals. In: Proceedings of the 2019 European Conference for Computing in Construction. pp. 341–350. European Council on Computing in Construction (2019), http://dx.doi.org/10.35490/ec3.2019.166

[7] Foundation, W.: Welcome to Wikidata. Available at https://www.wikidata.org/wiki/Wikidata:Main_Page 2022-12-27

[8] Tharani, K.: Much more than a mere technology: A systematic review of Wikidata in libraries. The Journal of Academic Librarianship **47**(2), 102326 (2021). https://doi.org/https://doi.org/10.1016/j.acalib.2021.102326, https://www.sciencedirect.com/science/article/pii/S0099133321000173

[9] Foundation, W.: Wikidata:Relation between properties in RDF and in Wikidata. Available at https://www.wikidata.org/wiki/Wikidata:Relation_between_properties_in_RDF_and_in_Wikidata 2022-12-27

[10] Erxleben, F., Günther, M., Krötzsch, M., Mendez, J., Vrandečić, D.: Introducing Wikidata to the linked data web. In: International semantic web conference. pp. 50–65. Springer (2014)

[11] Wikidata: Wikidata:data access. Available at https://www.wikidata.org/wiki/Wikidata:Data_access 2022-12-04 (2022)

[12] Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Trans. Database Syst. **34**(3) (sep 2009). https://doi.org/10.1145/1567274.1567278, https://doi.org/10.1145/1567274.1567278

[13] Lisena, P., Troncy, R.: Transforming the json output of sparql queries for linked data clients. In: Companion Proceedings of the The Web Conference 2018. p. 775–780. WWW '18, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE (2018). https://doi.org/10.1145/3184558.3188739, https://doi.org/10.1145/3184558.3188739

[14] Angele, K., Meitinger, M., Bußjäger, M., Föhl, S., Fensel, A.: GraphSPARQL: A GraphQL Interface for Linked Data. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. pp. 778–785 (2022)

[15] Taelman, R., Vander Sande, M., Verborgh, R.: GraphQL-LD: linked data querying with GraphQL. In: ISWC2018, the 17th International Semantic Web Conference. pp. 1–4 (2018)

[16] Dresslar, P.: Ready for GraphQL Sidebar: GraphQL vs. SPARQL? Available at https://medium.com/@peterdresslar/ready-for-graphql-sidebar-graphql-vs-sparql-4f2eb5246c12 2023-01-11 (2019)

Glossary

Q560 Is the ID for the item "Helium" in Wikidata. 5

Acronyms

BCP47 Best Current Practice 47. 4

IRI Internationalized Resource identifier. 3

SPARQL SPARQL Protocol and RDF Query Language. 6

URI Uniform Resource Identifier. 3

URL Uniform Resource Locator. 3

W3C World Wide Web Consortium. 4