FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Query by Graph

Bachelor Thesis

for the attainment of the academic degree

Bachelor of Science (B.Sc.)

in Computer Science

Friedrich Schiller University Jena

Faculty for Mathematics and Computer Science

submitted by

Friedrich Answin Daniel Motz

born on 15 July 2001 in Chemnitz, Germany

assessed by

Prof. Dr. Clemens Beckstein
M. Sc. Johannes Mitschunas

Jena, 12 January 2025

Abstract

This thesis introduces a Visual Query Builder for SPARQL queries, reducing the effort required to query Wikibase systems. At its core is a Visual Query Graph, which abstracts technical complexities of reified structures, such as Wikibase qualifiers, by leveraging a new labelled hyper-edge. Building on this concept, the newly implemented program features a robust, modular Rust backend and a Web frontend. Early testing by digital humanities students with FactGrid demonstrated the program's usefulness for constructing SPARQL queries.

Contents

Preface

This bachelor thesis represents the culmination of a journey fuelled by my commitment to making complex things more accessible. Along the way, I have been fortunate to receive invaluable support, guidance, and inspiration from several remarkable individuals.

First and foremost, I owe the genesis of this work to Olaf Simons. His blog post and the initiative FactGrid sparked my interest in exploring Visual Query Graphs and Wikibase, laying the foundation for this thesis.

I thank Lucas Werkmeister, whose expertise in the technical intricacies of Wikibase was indispensable. His guidance helped me navigate complexities I could not have overcome alone.

Special thanks go to Patrick Stahl for his contributions to implementing UI features. Your technical skills enriched the practical aspects of this work.

I owe the programs early public exposure to Clemens Beck. Thank you for testing the early preview of the program in your seminar and for providing crucial support and manpower to accelerate its development.

My deep gratitude goes to Clemens Beckstein and Johannes Mitschunas for their exceptional mentorship. Their wisdom, encouragement, and thoughtful feedback were instrumental in shaping this project and pushing it to its full potential.

I also wish to acknowledge the many friends, colleagues, and mentors whose support, guidance, and generosity of spirit have enriched this undertaking in countless ways.

But without you, Mom and Dad, I would never have had the opportunity to enjoy writing this thesis and to encounter so many interesting people and challenges. My deepest gratitude goes to you.

Each of you has played a vital role in bringing this thesis to fruition. Your support has made this journey not only intellectually rewarding but also personally meaningful.

To all of you, I extend my heartfelt gratitude.

1 Introduction

Over its thousands of years in existence, humanity has built an *infrastructure for knowledge*. It started out with stone tablets, evolved to hand-written papyrus books, libraries, the printing press and recently culminated in computer and the internet. Instead of using a library and asking a librarian, we usually consult "the internet" using a search engine – even for small questions. Now, in order to answer a question, the search engine needs to be able to treat the contents of a website in a semantically correct way, just like a human would. This is achieved using i. e. network analysis and techniques of natural language processing. However, what if the contents of websites could be semantically annotated by their creators?

This question lead to the inception of the Wikidata[1] initiative, among others. Their idea is to rewrite Wikipedia articles into very simple assertions using a specified vocabulary. These assertions consist of a subject, predicate and an object, in analogy to sentence structures in linguistics, where subjects and objects can refer to objects of our intuition and predicates define how they are related. These assertions are also referred to as **triples**. Another benefit of these triples is their ability to be represented as a graph (see Figure 1), where nodes represent subjects and objects (or e. g. Wikipedia articles), and edges represent predicates, also called properties or relationships. This triple structure allows to easily visualise the database's entries.

$$\text{Goethe (subject)} \xrightarrow{\text{educated at (predicate)}} \text{Leipzig (object)} \tag{A.1}$$

$$\text{Goethe} \xrightarrow{\text{place of birth}} \text{Frankfurt am Main} \tag{A.2}$$

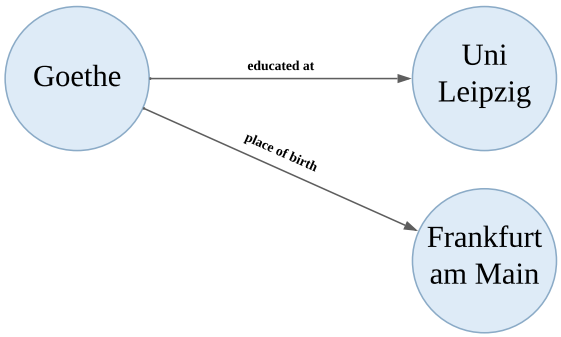


Figure 1: A graphical visualisation of the triples (Goethe, educated at, Leipzig) and (Goethe, place of birth, Frankfurt am Main) as a graph. Gothe is subject to both relationships, while edges represent predicates pointing to the the respective cities as objects.

Wikidata contains a very big set of such triples, posing the opportunity, that it could be used like a database and queried for information, just like classical relational databases. Such databases can be implemented using a framework

[1] https://www.wikidata.org

called Resource Description Framework (RDF) and are called **triplestore** or **RDF graph**. A resource can be any object of our intuition and these resources can be described using the syntax RDF offers. The vocabulary used to describe the resources, is specified or chosen by the users. Triplestores can be advantageous when the information collected is incomplete or might be enhanced later on. Applications using triplestores are inherently designed to handle the absence of data, as these systems lack a rigid schema. In contrast, relational databases enforce a strict schema that ensures every entry adheres to a predefined structure, enabling applications to consistently rely on well-organised and uniformly structured information.

The maximally flexible data model is what lead triplestores to become popular in the digital humanities. An initiative called FactGrid[2] hosts a triplestore specifically designed for historians, enabling them to make the data from their research publicly accessible. This poses the potential, that a user with knowledge of the specified vocabulary and conventions of the database, could get information about historical facts by writing an adequate query to the database. Furthermore, inferencing information about historical facts could be made a matter of, again, writing an adequate query.

1.1 Problem

Making use of a triplestore in a broader audience poses the challenge, that the technicalities of the database are exposed to its user. To populate and query the database, the users have to attend to the conventions of the vocabulary and the database engineers. The formalisation step is therefore being put into the hands of e. g. historians. Secondly, to adequately query a database, the user is forced to use the query language *SPARQL*, which requires technical knowledge.

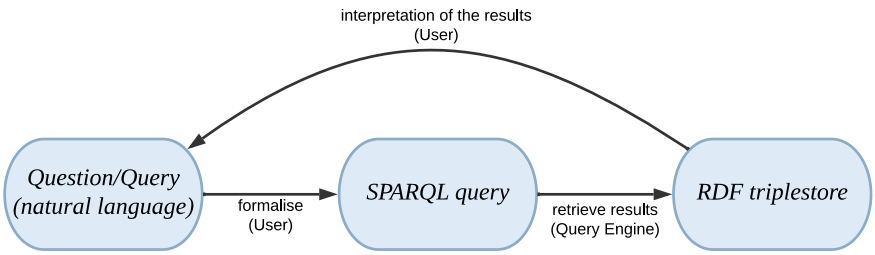


Figure 2: The process of getting a result from an RDF triplestore.

For example, a researcher might ask: “What professions did members of societies dedicated to advancements in the natural sciences in Jena hold?” There are many ways to interpret this question: Does the question refer to registered clubs, meaning a legal entity or does a regular’s table in a pub count? What does the term profession refer to? Is it the current *occupation* or the *trained* profession? Secondly, before starting to write a SPARQL query, the next step

[2] https://factgrid.de

is to ‘pre-formalise’ the question using the concise ‘subject, predicate, object’ syntax, to adequately captures the interpretation’s essence. This requires familiarity with the database’s modelling conventions. For example, a researcher could query for entities classified as clubs and ensure that these entities are also associated with ‘natural sciences’ through the predicate ‘interested in’. Alternatively, things related to ‘Natural research association’ through the predicate ‘instance of’ could be queried. Both options seem just, but in practice, only *one* returns results.

However, these initiatives want to reach a broader user base than the one likely to engage given these hurdles. It is unreasonable to expect users to navigate these steps without substantial training, a clear understanding of typical modelling practices, and in-depth knowledge of SPARQL language features.

```SPARQL
PREFIX fg: <https://database.factgrid.de/entity/>
PREFIX fgt: <https://database.factgrid.de/prop/direct/>
SELECT DISTINCT ?careerStatement WHERE {
  ?society fgt:P2 fg:Q266832 .
  ?society fgt:P83 fg:Q10391 .
  ?people fgt:P91 ?society .
  ?people fgt:P165 ?careerStatement .
}
```

Listing 1: A possible SPARQL query to the professions of members of societies for natural sciences in Jena from the database FactGrid.

1.2 Proposal

This work aims to lay the fundamentals for a program, which allows to build queries to an RDF triplestore using visual representation. The idea is, that since the *contents* of the RDF triplestore can be *visualised as a graph, so could the query* [1], [2]. Instead of writing a query in the database’s query language SPARQL, the user employs a visual query builder, which in turn generates the equivalent query.

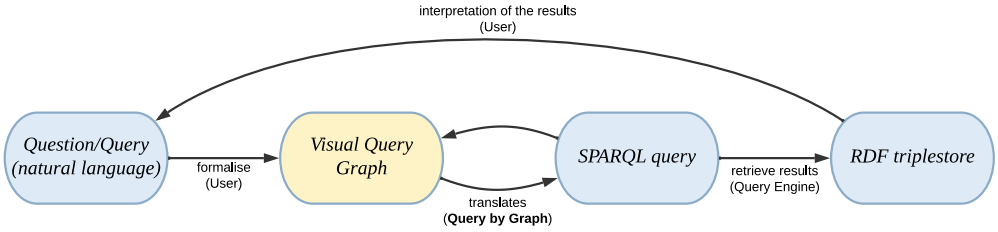


Figure 3: Methodology pipeline: How to get from a question in natural language to the result in an RDF database.

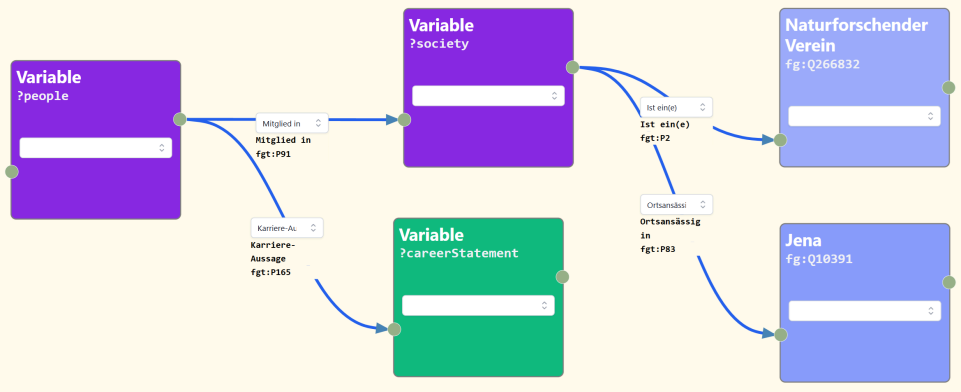


Figure 4: A screenshot of the Visual Query Graph which is generated to the query in Listing 1. Variables are shown in violet and things in light blue. Green nodes show which variables are part of the result set.

Creating a Visual Query Graph is similar to sketching: the user outlines the desired database structure and fills in variables for the desired results. The sketched graph is then automatically converted into a SPARQL query that adheres to all syntactical requirements. A result is retrieved from an RDF triplestore by finding the same graph structure as specified by the query. A variable will match any value in the RDF graph.

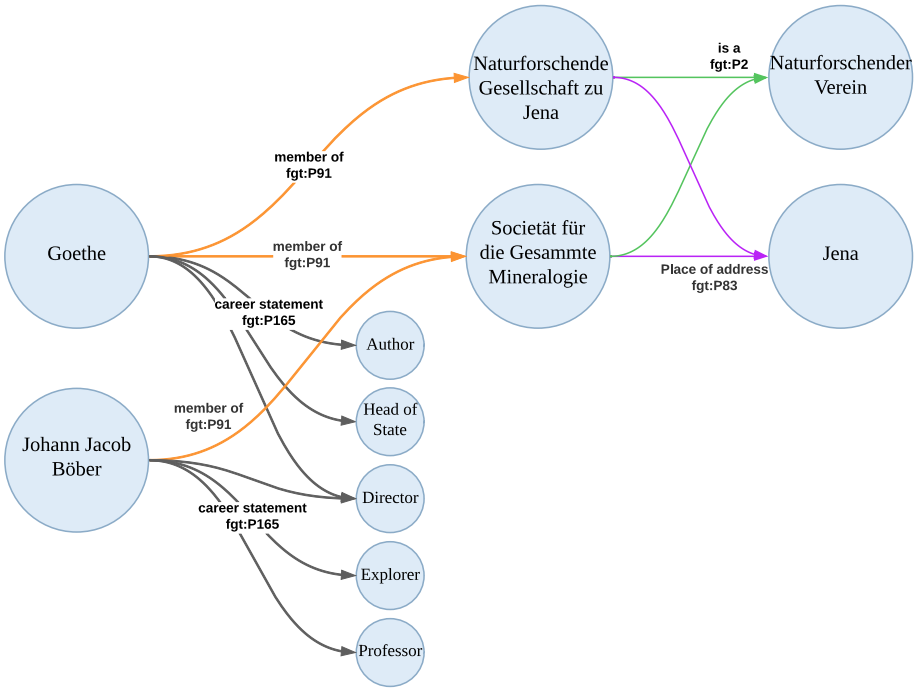


Figure 5: An exemplary RDF Graph against which the query from Figure 4 or equivalently Listing 1 is run.

The results of typical SPARQL queries on an RDF graph are presented as a table, with each column representing a requested variable. For this example, the table will have one column which includes all career statements associated with Goethe and Böber {Author, Head of State, Director, Explorer, Professor}.

This work aims to closely integrate with the triplestore software suite called

Wikibase[3], which is widely adopted[4]. Wikibase offers many very useful constructs, which, by their nature, require some technicalities to be represented using the triple syntax, e. g. further specifications of a property (which in Wikibase are called qualifiers). This work demonstrates that, beyond the standard triple syntax, such complex constructs can be represented as intuitive structures and queried using a Visual Query Graph, following [2]. To achieve this, it introduces the conventions of data modelling in Wikibase and explains their mapping to RDF syntax.

Query by Graph cannot fully eliminate the need for users to learn the conventions of an RDF triplestore. For instance, determining which subjects are available and what to expect is entirely dependent on the database's users and engineers, as is the naming of properties. However, with Query by Graph, querying an RDF graph becomes as simple as drawing a suitable stencil that mirrors the RDF graph's structure. The desired pattern is sketched, while any undefined elements are left as variables to be resolved during the query process. Using the editor's search fields, users can quickly adjust the meanings of nodes and edges, creating a workflow that feels more intuitive and similar to sketching a chain of thought.

Section 2 provides the necessary preliminaries, including the fundamentals of RDF, SPARQL, and the data model used in Wikibase. Section 3 delves into the principles of querying RDF graphs and the specific challenges posed by Wikibase's advanced constructs. Section 4 introduces the concept of Visual Query Graphs and how they are mapped to SPARQL queries. It furthermore discusses the implementation of the tool **Query by Graph**, which incorporates and realises a significant portion of the features conceptualised in this thesis. Section 5 discusses the implications, limitations, and potential extensions of the proposed approach.

[3] https://wikiba.se°

[4] e. g. Wikidata and FactGrid

2 Preliminaries

To define the tasks of Query by Graph, it is essential to discuss Wikibase's data modelling conventions, the formal definitions of Wikibase's special constructs, their mapping to the Resource Description Framework (RDF), the RDF itself and the syntax of the query language for RDF, SPARQL. The most commonly used SPARQL queries for retrieving information are SPARQL-SELECT queries, which are the primary focus of this work. SPARQL-SELECT queries function like stencils that describe a triple pattern, which is applied across an RDF graph until a matching pattern is found. For each match in the RDF graph, the corresponding variable assignments are returned as a result set. The idea is that the Visual Query Graph will represent the same stencil as the SPARQL-SELECT query.

Certain patterns in the RDF graph of Wikibase exist solely for technical reasons and are not intuitive to users without an understanding of the underlying necessities. For example, this includes relationships involving multiple objects. These patterns are limited in scope and are defined within the Wikibase data model, providing an opportunity to develop an intuitive representation for them in the Visual Query Graph. During query generation from the Visual Query Graph, these intuitive representations are translated into technically accurate constructs, ensuring they can be queried successfully.

2.1 Resource Description Framework

To introduce the Wikibase data model and its mapping to the Resource Description Framework (RDF), it is essential to first understand the terminology of RDF.

2.1.1 Internationalised Resource Identifier

Internationalised Resource Identifiers (IRIs) [RFC3987°] are a superset of Uniform Resource Identifiers (URIs) [RFC3986°], for example `http://database.factgrid.de/entity/Q409` and `https://database.factgrid.de/prop/direct/P160`. Their purpose is to unambiguously **refer to a resource** across all triplestores (or the WWW). The resource an IRI points at is called **referent** [3].

Remark. IRIs can largely be treated as URIs, as they are interchangeable through conversion. Their primary purpose is to **identify the entity being referenced within a specific triplestore or Wikibase instance**. Since the technical details are not directly relevant to this work, I will refer readers to the referenced RFCs for further information.

2.1.2 Prefixing

RDF allows to define a **prefix**, which acts as an **abbreviation of an IRI**. For example, let `wd` be a prefix with the value `http://www.wikidata.org/entity/`.

Then, the IRI `http://www.wikidata.org/entity/Q5879` can be rewritten using this prefix as `wd:Q5879`. The part after the colon is called **local name** and is essentially a string restricted to alphanumerical characters [3]. The term "prefix" will also be used to describe specific prefixes in the Wikibase data model: Each Wikibase instance defines a set of prefixes and data modelling conventions around them.

2.1.3 Literals

A **literal** in an RDF graph can be used to express values such as strings, dates and numbers. It essentially consists of two elements[5]:

1. a **lexical form**, which is a Unicode string,
2. a **data type IRI**, which defines the mapping from the lexical form to the literal value in the user representation.

2.1.4 Blank nodes

RDF specifies **blank nodes**, which do not have an IRI nor a literal assigned to them. Most common syntax expresses blank nodes with a "blank IRI" denoted by an underscore followed by a local name, e. g. `_:implicit1`. The specification [3] and the current version of its successor [4] do not comment on the structure of a blank node: "Otherwise, the set of possible blank nodes is arbitrary." [3]. It only specifies, that **the set of blank nodes is disjunct from all literals and IRIs**. It furthermore specifies, that: "Blank nodes in graph patterns [for SPARQL queries] **act as variables**, not as references to specific blank nodes in the data being queried" [5]. This means, that variables can be used to query blank nodes and are treated the same way by the query engine.

2.1.5 RDF Triple and RDF Graph

To establish a concise notation for subsequent definitions, this work introduces specific sets to be used throughout. The set of all IRIs is represented by $\boldsymbol{I}$, the set of all blank nodes by $\boldsymbol{B}$, and the set of literals by $\boldsymbol{L}$. The set of all valid RDF terms is defined as $\boldsymbol{T} \coloneqq \boldsymbol{I} \cup \boldsymbol{L} \cup \boldsymbol{B}$.

Definition 2.1. Let $\boldsymbol{s} \in \boldsymbol{I} \cup \boldsymbol{B}$ be a subject, $\boldsymbol{p} \in \boldsymbol{I}$ a predicate and $\boldsymbol{o} \in \boldsymbol{T}$ an object.

Then, following [3], an **RDF triple** or simply a **triple**, is defined as

$$(\boldsymbol{s}, \boldsymbol{p}, \boldsymbol{o}). \tag{B}$$

Definition 2.2. An **RDF graph** is a set of RDF triples. An RDF triple is said to be asserted in an RDF graph if it is an element of the RDF graph [4].

[5] The specifications and the new proposal for RDF allow for more elements for language-tagging [3], [4], however, they are not relevant to this work.

2.2 Data Model in Wikibase

Wikibase is one of the most widely used softwares for community knowledge bases, with the most prominent instance, **Wikidata**[6], storing ~115 million data items. Wikibase has its own internal structure and conventions for naming and modelling entities and concepts. These internals are in turn mapped to an expression in RDF syntax [6]. This invertible mapping permits the use of *RDF terminology to refer to structures within Wikibase* and most notably *the use of the SPARQL query language for information retrieval.* This specific data model of Wikibase is particularly noteworthy due to its widespread use and substantial influence on other initiatives, driven by its sheer scale. For example, DBpedia will make use of Wikidata resources [7].

In Wikibase, a **thing** is referred to as an **item** and assigned a unique **Q-Number** within a Wikibase instance. Any **predicate** is called **property** and assigned a unique **P-Number**. A statement in Wikibase puts an item in relation to another item using a property.

Example 2.3. Suppose a user wants to enhance an entry in Wikidata for a person called "Johann Wolfgang von Goethe". Goethe is modelled as an item with the Q-number `Q5879` and wants to add the statement, that Goethe was "educated at" (P-number `P69`) the "University of Leipzig" (Q-number `Q154804`). Using the user interface, the user edits the entry for Goethe and fills the fields "property" and "object" with `P69` and `Q154804`. The triple representation in an RDF graph would be very similar:

$$\text{Johann Wolfgang von Goethe} \xrightarrow{\text{educated at}} \text{University of Leipzig}, \quad \text{or} \tag{C.1}$$

$$\text{Q5879} \xrightarrow{\text{P69}} \text{Q154804}. \tag{C.2}$$

Most real-world relationships might present to be more complex than something one would want to model in a single triple. For example, one may want to express that "Goethe" was educated at the "University of Leipzig" from 3 October 1765 to 28 August 1768. Wikibase represents it as a hierarchical structure, with "educated at" as the primary property and the others arranged beneath it. In the Wikibase context, a statement specifying another relationship is called a **qualifier**.

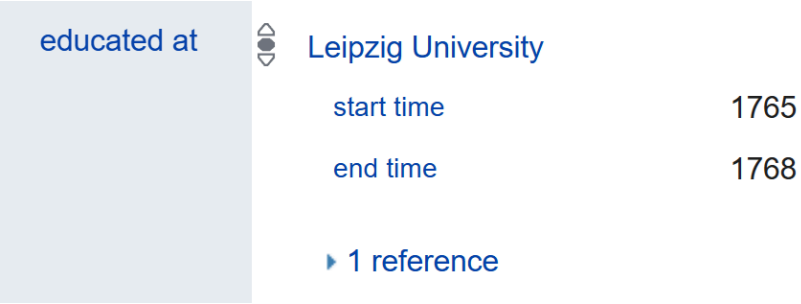


Figure 6: Presentation of an qualified relationship in the software Wikibase.

[6] http://wikidata.org° — an initiative for a free community knowledge base

One possibility is to let relationships have more than two operands, i. e. increase the arity by one for each additional parameter. “Educated at” would then be called “educated at (·) from (·) to (·)”. Another way using the triple syntax is to create an implicit object, that assists in modelling the relationship using an **implicit** or **blank node** to describe a new concept; a human might be inclined to give it a name, e. g. “educated at for a certain time”. This act is also called **reification** (objectification of a fact). The following triples exemplify such an implicit relationship, called a **qualified statement**:

$$\text{Goethe} \xrightarrow{\text{educated at}} \text{Uni Leipzig}, \tag{D.1}$$

$$\text{Goethe} \xrightarrow{\text{educated at}} \text{Implicit1}, \tag{D.2}$$

$$\text{Implicit1} \xrightarrow{\text{location}} \text{Uni Leipzig}, \tag{D.3}$$

$$\text{Implicit1} \xrightarrow{\text{started at}} \text{3.10.1765}, \tag{D.4}$$

$$\text{Implicit1} \xrightarrow{\text{ended at}} \text{28.08.1768}. \tag{D.5}$$

The statements of Eq. D.4 and Eq. D.5 are called **qualifiers**.

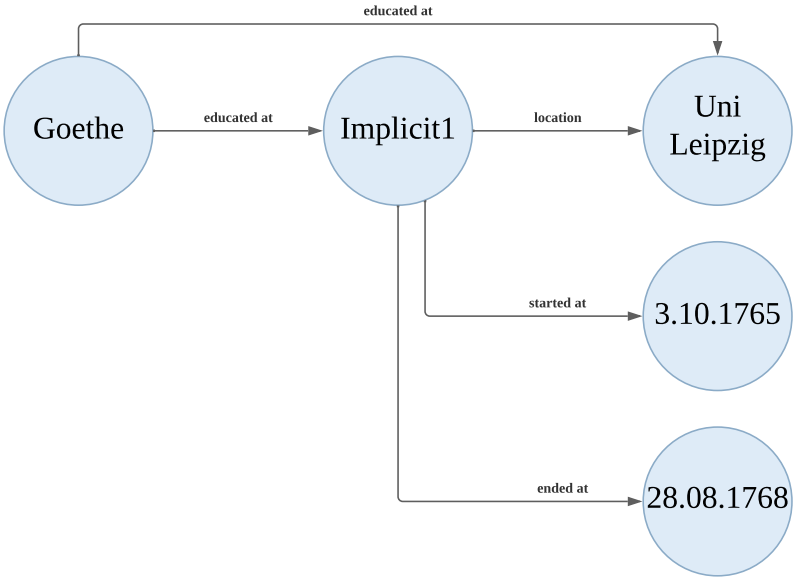


Figure 7: Graphical visualisation of a qualified statement using natural language descriptors.

Wikibase instances define IRI-prefixes for things of the same kind. This allows to think of them as namespaces for categories defined within the Wikibase data model. Since this work treats the set of all possible Wikibase instances where the IRIs use different domain names from e. g. `wikidata.org`, they can be thought of as variables for the instance-specific prefix. For convenience, these variables will be denoted by the prefix names followed by a colon (`wd:`, `p:`, `pq:`, `wdt:` and so on) defined by Wikidata (see Listing 2). For the matters of this work, only IRIs which can be written as the concatenation of the prefix with a local name (an alphanumerical string) are considered to be an *element of the namespace*, e. g. `http://www.wikidata.org/prop/P1234` is an element of

the namespace `p:` but `http://www.wikidata.org/prop/something/else/P1234` is not an element. Furthermore, the mapping of the Wikibase data model to RDF syntax specifies that these namespaces can only be used in triples (or edges, for that matter) that connect specific namespaces. *The use of these namespaces is therefore restricted.* For example, an edge with a referent in the namespace `p:` can only have sources in the namespace `wd:` and only targets in the namespace `wds:`. Figure 8 is an illustration taken from the Wikibase documentation on RDF mapping, which gives an overview of these restrictions.

```sparql
PREFIX p: <http://www.wikidata.org/prop/>
PREFIX pq: <http://www.wikidata.org/prop/qualifier/>
PREFIX pqv: <http://www.wikidata.org/prop/qualifier/value/>
PREFIX ps: <http://www.wikidata.org/prop/statement/>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wds: <http://www.wikidata.org/entity/statement/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wdv: <http://www.wikidata.org/value/>
```

Listing 2: An excerpt of customary IRI prefixes defined by Wikidata.

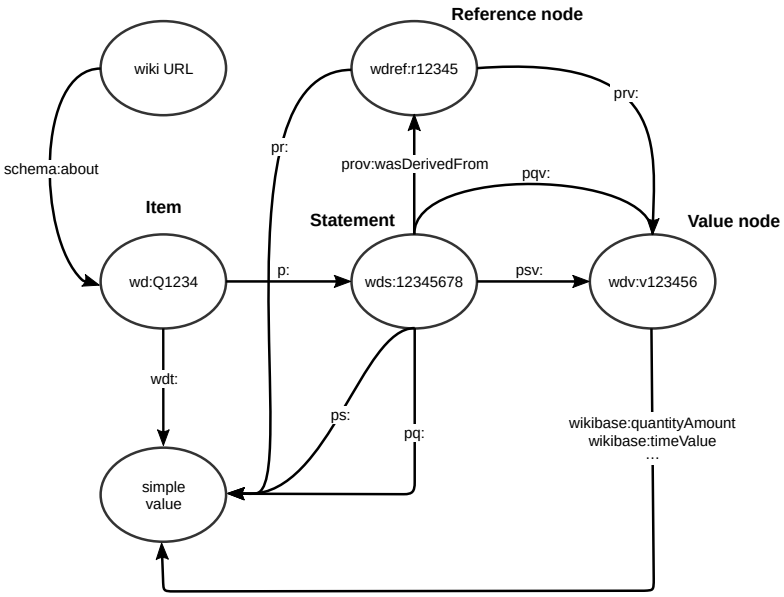


Figure 8: An overview of restrictions for the use of namespaces in Wikibase [8]. The labels of the nodes and edges act as placeholders for specific IRIs, whose referents are within the namespace indicated by the label.

For further use, a subset of these namespaces will be denoted by abbreviations. Let Σ be a valid alphabet for local names and Σ^* its Kleene closure. Let $f_p, f_q, f_s \in I$ be *distinct* IRIs. In analogy to the Wikibase data model, f_p will denote the prefix for the namespace for **p**roperties `p:`, f_q for **q**ualifying properties `pq:` and f_s for **s**tatements `ps:`.

3 Querying

With the structure of the data to be queried now clearly defined, the next step involves formulating a corresponding SPARQL query and creating a query yielding the same results using a Visual Query Graph.

Constructing a query for an RDF graph can be viewed as creating a subgraph — a set of RDF triples. In addition to valid RDF terms such as IRIs, blank nodes, and literals, now variables can be inserted at any position in the triple, instead of an RDF term. Each variable is distinct from all others and can be placed in multiple positions within the query graph. The database's query engine will try to find the same structure in the RDF graph and returns the RDF Terms which overlapped with a variable. This *matching* is essentially the process of querying an RDF graph. Among other features, such stencils can be written using the RDF query language *SPARQL*, where this query graph or stencil is referred to as *Basic Graph Pattern*.

3.1 SPARQL Protocol and RDF Query Language

The acronym *SPARQL* is recursive and stands for **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage and is part of the Resource Description Framework recommendation. It is considered to be a *graph based* query language. The definitions of the following section are an excerpt from the *Formal Definition of the SPARQL query language* [9]. All relevant aspects of the formal definition are clarified in this work. Readers interested in further details are encouraged to consult the documentation directly.

This work focuses on a specific subset of SPARQL queries, specifically SPARQL-SELECT queries. SELECT queries can include additional components, such as value constraints, which restrict permissible variable assignments in the results. For instance, a constraint ensuring that e. g. an event occurred before 1900 would be expressed as `FILTER(?year < 1900)`. Such language features are not yet specified in the Visual Query Graph.

Other types of SPARQL queries also exist, such as `ASK` and `DESCRIBE`, which differ in the structures they return. These SPARQL query types and also the data manipulation language for triplestores will not be dealt with in this work. They are detailed in the SPARQL specification [5].

For further use, the set of all variables is from now on denoted by $\boldsymbol{V}$. Following the syntax of SPARQL, variables in examples will be denoted with a leading question mark `?` followed by an alphanumerical word.

Definition 3.1. A **Basic Graph Pattern (BGP)** is a **subset** of SPARQL triple patterns [9]

$$(T \cup V) \times (I \cup V) \times (T \cup V). \tag{E}$$

Example 3.2. A valid Basic Graph Pattern following the query in Listing 1 and visualised in Figure 4 would be

$$\{(?\text{society}, \text{instance-of}, \text{natural research association}),\\ (?\text{society}, \text{located-in}, \text{Jena}),\\ (?\text{people}, \text{member-of}, ?\text{society}),\\ (?\text{people}, \text{career-statement}, ?\text{careerStatement})\}. \qquad (F)$$

Definition 3.3. In essence, a **Graph Pattern** can contain multiple and optional **Basic Graph Patterns**. This work concentrates on queries which only contain one Basic Graph Pattern.

Definition 3.4. A **SPARQL-SELECT query** is a special SPARQL query, which consists of a Graph Pattern, a target RDF graph and a result form. The so-called result form specifies how the result of a SPARQL query looks like. In the case of SELECT queries it is a projection to the valid variable assignments for the given Graph Pattern. Alternative result forms include `ASK` and `DESCRIBE`. The return tuple is determined by the query's projection statement, which specifies the subset of variables from the query to be included. The query results can be ordered using solution modifiers e. g. `DISTINCT`, `LIMIT` or `ORDER` (in analogy to SQL).

Remark. Since a basic graph pattern can have any RDF Term as a subject, this implies, that a SPARQL query can query for a triple, which has a literal as its subject. An RDF graph however cannot have a triple with a literal as a subject.

Writing SPARQL queries is pretty straight-forward: The wanted structure is expressed in terms of the query language, and the unknown parts are replaced by variables. Say the user wants to know which universities Goethe went to. The matching query would look like Listing 3. IRIs are enclosed within angle brackets.

```SPARQL
PREFIX wd: <http://www.wikidata.org/entity/> # for brevity
PREFIX wdt: <http://www.wikidata.org/prop/direct/> # for brevity

SELECT # result form
  ?institution # projection statement
WHERE
  { # graph pattern, in this case a basic graph pattern ...
    wd:Q5879 wdt:P69 ?institution . # ... with one entry
    # Johann Wolfgang von Goethe -- [educated at] -> Variable
  }
```

Listing 3: A SPARQL query to determine which educational institutions Goethe visited. Currently, the valid results are `wd:Q154804` (University of Leipzig) and `wd:Q157575` (University of Strasbourg). The structural components from Definition 3.4 are highlighted with comments.

In order to query a BGP containing a blank node, a query has to specify a variable at the blank node's position. There are other syntactical structures to

express blank nodes, which are however semantically equal to using a variable [9].

3.2 Qualifiers

The term qualifier is not clearly defined in the documentation around Wikibase [10], [11]. An achievement of this work is the dissemination of the terminology, in order to create adequate queries for these structures. Listing 4 shows an exemplary query for qualifiers. By incorporating the namespace conventions of Wikibase as shown in Figure 8, it becomes evident that the variable `?implicit1` matches a node within the `wds:` namespace. The values of these nodes are however irrelevant for querying and therefore to this work. Therefore, the choice was made to ignore this implementation detail in the following definitions, and treat them as blank nodes. As already mentioned in Section 2.1.4, querying using blank nodes and variables yields the same results.

```SPARQL
SELECT ?startDate WHERE {
    wd:Q5879 p:P69 _:implicit1 .     # this and the following line
    _:implicit1 ps:P69 wd:Q154804 .  # specify the narrow qualified
    statement
    _:implicit1 pq:P580 ?startDate . # and this queries the
    qualifier's referent
}
```

Listing 4: A query to fetch the start date of Goethe's education at the University of Leipzig using the prefixes posted in Listing 2.

Obeying the Wikibase data model and its namespace conventions, a **qualifier** or **qualifier edge** is an edge pointing from an element of the namespace `wds:` to an element of any namespace using a predicate in the `pq:` namespace. The **value of a qualifier** is the target node of this edge.

Definition 3.5. Let Σ be a valid alphabet for local names and Σ^* its Kleene closure. Any RDF triple with a predicate of the form $f_p u$ with $u \in \Sigma^*$ is a **qualifier**.

Definition 3.6. Let Σ be a valid alphabet for local names and Σ^* its Kleene closure. Let $s \in I$ be a subject, $b \in B$ a blank node, $o, o' \in I \cup L$ objects, which are all elements of G. Then, any subgraph $G_{\mathrm{QSn}} \subset G$ with

$$G_{\mathrm{QSn}} := \{(s, f_p u, b), (b, f_s u, o)\} \tag{G}$$

is called **Qualified Statement in the narrow sense**, and o' is called **Qualifier Value**, where $u, u' \in \Sigma^*$. Furthermore $G_Q \subset G$,

$$G_Q := \{(b, f_q u', o')\}, \tag{H}$$

is called qualifier to the Qualified Statement in the narrower sense. A **Qualified Statement in the broader sense** is the union of all qualified statements in the narrower sense to a specific blank node.

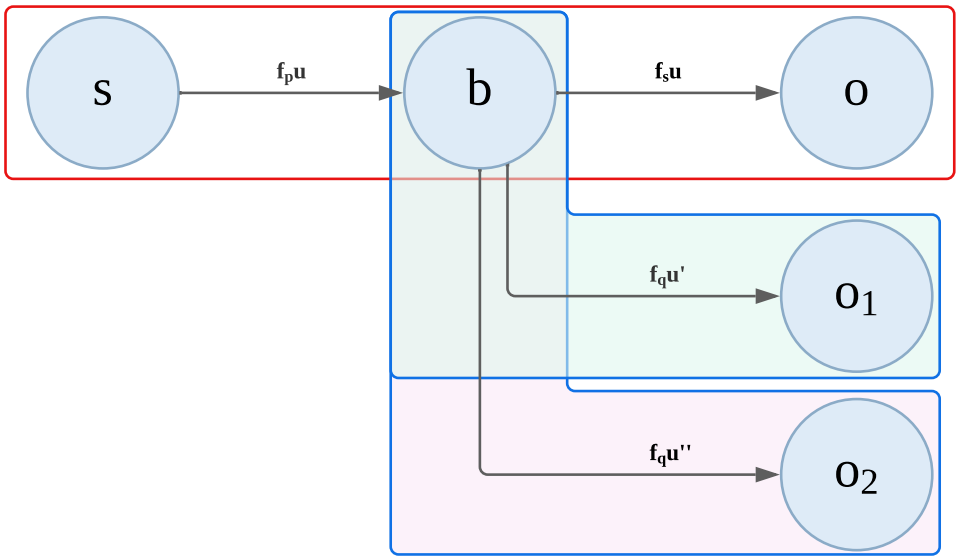


Figure 9: A visualisation of a qualified statement in the broader sense with two qualifiers using the terms introduced in Definition 3.6 and u, u', u'' are local names. The red box indicates the qualified relationship, the green box one qualifier and the violet box the other qualifier.

4 Mapping

To establish a semantic-preserving mapping between Visual Query Graphs and SPARQL-SELECT queries — meaning it yields the same results in both formalisms – a formal specification for Visual Query Graphs is introduced. Following this, the transformation function from a Visual Query Graph to SPARQL is defined. Lastly, the implementation of these functions is analysed and discussed.

4.1 Visual Query Graphs and Basic Graph Patterns

The so far introduced structures include Basic Graph Patterns in SPARQL queries and RDF triples. While Basic Graph Patterns are used to describe stencils to be queried against RDF triples, the goal of Visual Query Graphs is to eliminate the need to manually model reified structures using blank nodes. Other literature [1] uses the term Visual Query Graph to refer to a Basic Graph Pattern without Blank Nodes. This work uses the same term to define a graph with multi-edges, which consist of the same triples as the Basic Graph Pattern would, but with the exception, that any qualifier structures are replaced with a multi-edge, including all statements of the qualified statement in the broader sense.

In order to build a Visual Query Graph, we need special edges which involve all nodes of a qualified statement in the broader sense. This can be done using a hypergraph. A qualifier will be a hyperedge consisting between at least three nodes using at least two edges. All Basic Graph Patterns which are not qualified statements in the broader sense will be copied to the Visual Query Graph without any changes. All qualified statements will be exchanged for a hyper-edge, where the blank node is removed and the edges will be rebuilt using one directed hyper-edge. The visualisation of the hyper-edge in the VQG can be seen in Figure 10.

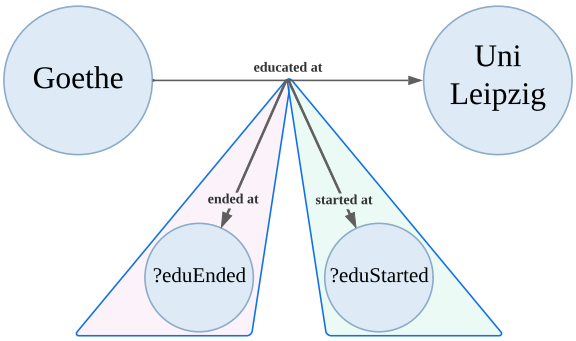


Figure 10: Visual Query Graph with two Qualifiers. The equivalent SPARQL query should return two qualifier values. The qualifiers are highlighted using a violet and a green box.

Definition 4.1. Let $G = (X, E)$ be a hypergraph. A hyperedge $(U, V) \in E$ is defined as a 2-tuple, where $U \in X$ are the source nodes and $V \in X$ are the

target nodes. A **labelled hyperedge** (s, V) is a simplified hyperedge, between a source node n and a set of labelled target nodes $(p, o) \in V$ with label p and target node o.

Definition 4.2. Let Σ be a valid alphabet for local names and Σ^* its Kleene closure. Let $G \in (T \cup V) \times (I \cup V) \times (T \cup V)$ be a Basic Graph Pattern. Furthermore, let $N \subset I \cup L \cup V$ be a set of nodes, $E \subset N \times (I \cup V) \times N$ a set of edges, and $E_q \subset N \times \mathcal{P}\big(\{f_s u \mid u \in \Sigma^*\} \cup \{f_q u \mid u \in \Sigma^*\} \times N\big)$ a set of labelled hyper-edges for qualified statements, where $\mathcal{P}(X)$ denotes the power-set of a set X. Then the corresponding **Visual Query Graph** $G_q = \big(N, E, E_q\big)$ is a special directed hypergraph to a Basic Graph Pattern G and constructed as follows:

1. Add all nodes the set of nodes N of G_q.
2. Copy all elements of G to the the set E of G_q, but remove all Qualified Statements.
3. For each Qualified Statement G_{QS} in the broader sense in G, create one labled hyperedge e_q in E_q as follows:
   1. From the triples of the Qualified Statement in the narrower sense Q_{QSn}, add a tuple which omits the blank node and goes to the object: $Q_{\text{QSn}} \subset Q_{\text{QS}}$, $G_{\text{QSn}} = \big\{(s, f_p u, b), (b, f_s u, o)\big\}, u \in \Sigma^*, o \in N$. The labelled hyperedge e_q is then $(s, \{(f_s u, o)\})$ and
   2. for each qualifier to G_{QS} of the form $\big(b, f_q u', o'\big), u' \in \Sigma^*, o' \in N$ add a tuple $\big(f_q u', o'\big)$ to the targets.

Example 4.3. The Visual Query Graph $G_q = \big(N, E, E_q\big)$ illustrated in Figure 11 would have the following sets (for brevity, the prefix `wd` for the items `Q5879` and `Q154804` are omitted):

$$N := \{(\text{Q5879}, \text{Q154804}, \text{?eduEnded}, \text{?eduStarted})\}, \tag{I.1}$$

$$E := \{\}, \tag{I.2}$$

$$E_q := \{(\text{Q5879}, \{(\text{ps:P69}, \text{Q154804}), (\text{pq:582}, \text{?eduEnded}), (\text{pq:580}, \text{?eduStarted})\}) \tag{I.3}$$

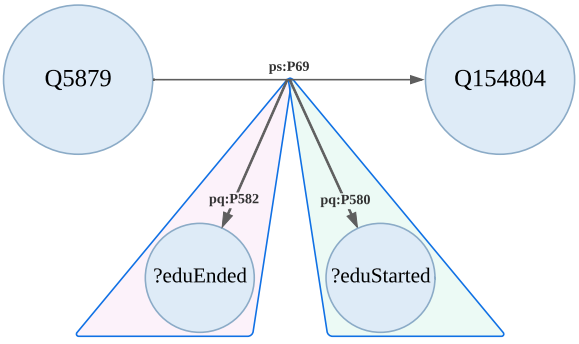


Figure 11: Visual Query Graph with two qualifiers using the accurate Wikibase prefixes.

To construct a valid BGP from a VQG, all regular edges are copied, and for each hyperedge the BGP receives at least three edges: first, the qualified statement in the narrower sense is added, as an edge from the source of the hyperedge to a newly inserted blank node. Second, for each of the hyperedge's target tuples t, an edge in the BGP is added from the same blank node to node in t using the the label in t.

4.2 Specification

This section outlines how the Visual Query Graph and hyperedges can be built using the visual query interface. Following [1], the VQG is *constructed* using the *Visual Query Language (VQL)*, consisting of four algebraic operators, which will correspond to atomic user interactions in the user interface (see Table 1).

User Interaction
Adding a variable node
Adding a literal node
Adding a directed edge
Adding a qualifier to an edge

Table 1: Operations in the VQL

The purpose of the hyperedge in the VQG is to express the association between the relationships recorded within it. When rendering the graph using the display library, however, the edge between the subject and object should be hierarchically emphasised, with qualifier edges subordinated, as illustrated in Figure 11. Consequently, in the data structure of the VQG, a qualifier should be considered as a property of the regular edge connecting two nodes. Therefore, the addition of a qualifier is always in reference to an existing edge and will be modelled as such.

Using this new VQG and VQL, we can now create an intuitive visualisation (see Figure 10) as motivated by [2]. The mapping from Visual Query Graphs to Basic Graph Patterns follows the definition of the construction, but with some simplifications.

4.3 Implementation

The goal of this work is to create two program parts, the hope being that i. e. the backend can be reused by other projects:

1. the *visual query building interface* (forthon called **frontend**) and
2. the *translator between VQG and SPARQL* (forthon called **backend**).

The most important aspects for the choice of software and UX design were usability, maintainability and reusability. The aim is to lay the basis for a

software, which can be applied in day-to-day use as an "almost-no-code" query builder.

4.3.1 Architecture

Given RDF's predominant use in web contexts, opting for a web application was a natural choice. The backend was designed to be both explainable and traceable. While several functional programming languages are well-suited for this purpose, Rust[7] emerged as the preferred option due to its ability to compile to WebAssembly[8], enabling native execution in a browser.

The **frontend** was developed using TypeScript, Vite, Vue3, ReteJS, and TailwindCSS, all licensed under the MIT license. Its purpose is to allow the user to

1. build a VQG, and edit it from the SPARQL code editor,
2. searching for items and properties in arbitrary Wikibase instances using the API,
3. display meta-information on items and properties,
4. configure Wikibase data sources and
5. handle all data source specific tasks (such as enriching entities with information from a the Wikibase API).

This approach presents a significant advantage over e. g. traditional server-client architectures by combining extensibility, efficiency, and formal precision. Rust's algebraic type system plays a central role in ensuring robustness, as it enforces the consideration of all possible cases, leaving no room for omissions. This guarantees a high level of reliability in the system's design. Moreover, since the entire computation occurs on the client side, the performance benefits are substantial, with rapid execution speeds. The use of WebAssembly further enhances this efficiency by providing a highly optimised runtime environment. Additionally, the modular design of the architecture simplifies the process of integrating new formats or types. By defining and handling these changes only at the interface boundaries, the system avoids unintended side effects, ensuring a predictable and maintainable implementation. This combination of features makes the architecture both robust and adaptable to evolving requirements.

The **Wikibase data sources** are configured by the user and stored in the browser's local storage. Following the conventions of Wikibase, the choice was made to only allow one prefix for items and one for properties. In the context of Visual Query Graphs it only makes sense that the item prefixes point directly to the item, e. g. `wd` for Wikidata, and the property prefixes to the property value, e. g. `wdt`.

The **backend** is designed to parse SPARQL queries into Query Graphs and convert them back. It utilises the `spargebra`[9] library for parsing SPARQL queries, though this library is still under development. Verifying the correctness of the

[7] http://www.rust-lang.org°
[8] http://webassembly.org°
[9] https://docs.rs/spargebra/latest/spargebra/°

parser lies outside the scope of this work. Nevertheless, it was confirmed that the parser produced correct results for randomly generated SPARQL-SELECT queries with BGPs.

To ensure compatibility between the backend and frontend, both use the exact same types with equivalent data types in their environments. This ensures the correct exchange of data between both representations.

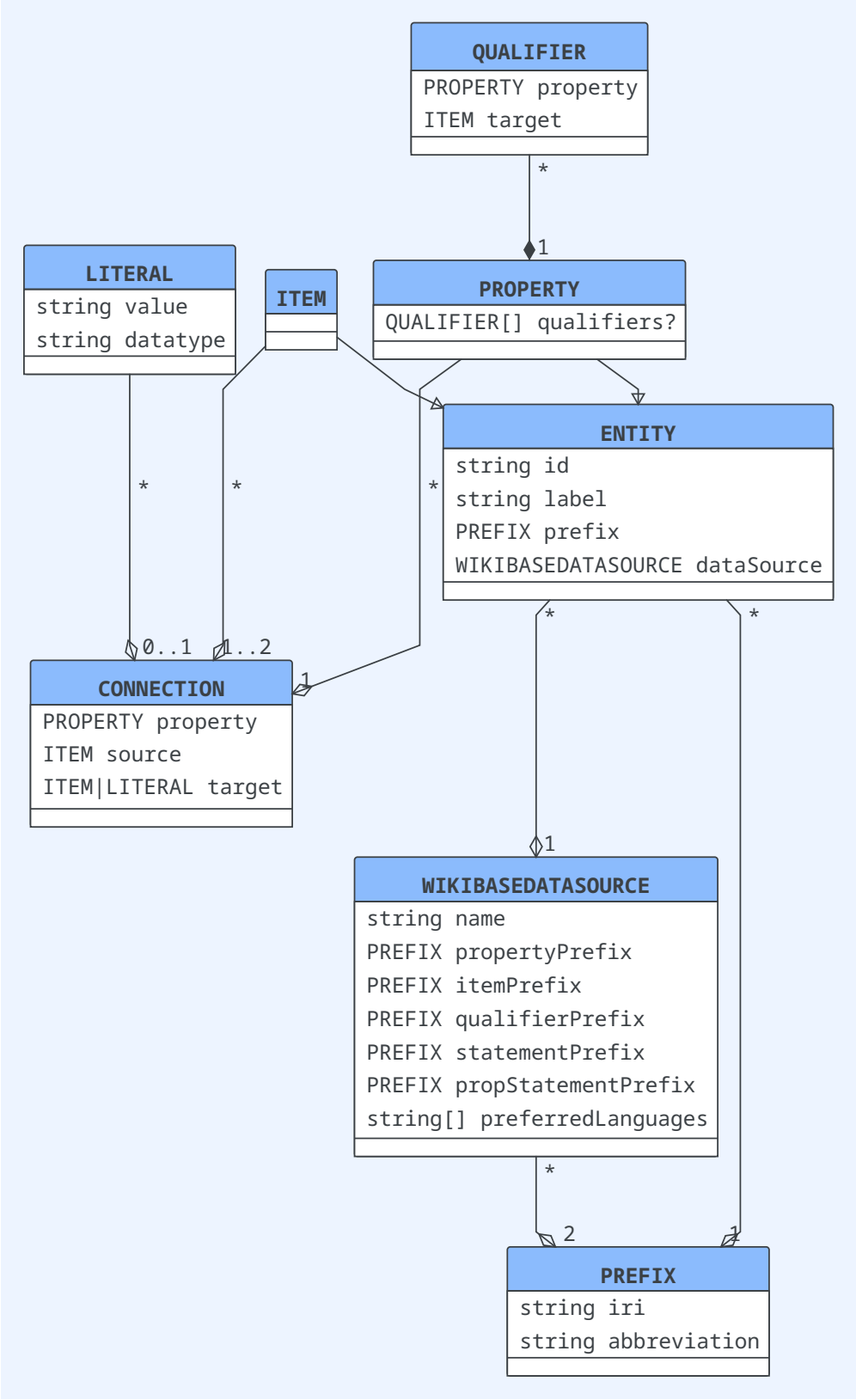
The VQG is exported in the form of an edge list from the frontend to the backend. The elements of the edge list are triples, corresponding to BGPs, and each entry of the triple is a literal, variable or IRI. The BGPs in turn are mapped to a SPARQL-SELECT query with all variables from the VQG added to the projection.

4.3.2 Visual Query Graphs in the Implementation

The following listing shows a UML diagram containing the key data types. The class diagram is so complex because of the support for multiple Wikibase instances, which comes with the necessity to store meta-information about the entities in a Visual Query Graph.

The key challenge in mapping a VQG to a BGP is to insert the correct edges for qualified statement in the broader sense. The frontend passes the qualifiers “as is” to the backend, which uses the prefixes specified in the Wikibase data source configuration to insert the adequate edges. This BGP is in turn used to generate a SPARQL-SELECT query.

The inverse operation requires pattern matching for qualified-statements in the broader sense, which is also done in the backend. The instantiated objects are passed to the frontend, which sets its current Visual Query Graph with the incoming data and redraws the visualisation.



Listing 5: Key Data Types for the translation between VQG and SPARQL.

5 Discussion

5.1 Evaluation

The development of Query by Graph represents a significant contribution to enhancing the usability of Wikibase instances, particularly in the context of digital humanities. This work will be part of the ongoing DFG°-funded *HisQu* project in collaboration with the MEPHisto group. Therefore, the focus in this work lay on establishing a robust, extensible, and modular platform. During this thesis, the tool received preliminary testing in a digital humanities seminar, supported by Patrick Stahl's contributions to the development of the user interface components. All changes to the code base with attribution are documented in the repository's° version history. The program can be used in a web browser and is accessible at https://quebyg.daniel-motz.de/°.

Central to this work is the precise analysis of Wikibase conventions and the introduction of well-defined terminology, including the novel concepts of Qualifier-Centric Representations and Hyper-Edges for Blank Nodes.

These novelties are an advantage over existing approaches, which also use the term Visual Query Graph [1]. Existing documentation often lacks terminological clarity, complicating the onboarding process for new users. This work addresses these gaps and systematically presents the equivalences between Basic Graph Patterns and Visual Query Graphs in an accessible manner.

Queries involving qualifiers can easily fail due to minor syntactical errors or wrong prefixes, which result in empty results without clear feedback. By explicitly incorporating these RDF constructs into the Visual Query Graph, this work mitigates these issues, thereby making the querying of qualifiers accessible for many users in the first place.

The implementation of the proposed Visual Query Graph and Mapper called Query by Graph, introduces an enhanced version of a visual query interface compared to [1], [12] that prioritises user-friendly design. *The current implementation does not fully include qualified statements.* The backend, implemented in Rust running natively in the browser, delivers outstanding responsiveness and robustness. Unlike existing tools using a graph representation [1], [12], it supports the backwards translation of SPARQL queries into Visual Query Graphs, enabling bidirectional interaction. This dual representation simplifies query construction. Observing the step-by-step construction of a Visual Query Graph can serve as an effective aid in understanding and learning the SPARQL syntax.

The tool also supports dynamic configuration and switching between multiple Wikibase instances, enabling users to query multiple data sources seamlessly.

Other approaches like RDF Explorer [1] can be adapted to different data sources too, they require recompilation and source code modifications.

A preliminary user study with digital humanities students demonstrated that the tool could be effectively employed with minimal training, a finding consistent with prior research. The student's task was to build queries given in natural language and to test, whether they yielded the expected result. However, further comprehensive studies are necessary to validate its long-term usability and effectiveness.

5.2 Future Prospects and Limitations

Future work will explore the integration of ontology-driven query snippets, following [13]. Unlike [13], which mandates an ontology for every query, Query by Graph allows users to derive query fragments directly from ontology snippets, providing a more flexible and intuitive mechanism for constructing complex queries.

Currently, Query by Graph supports only SPARQL-SELECT queries with a single Basic Graph Pattern. Future enhancements could include the visualisation of optional graph patterns, value constraints (e. g., `FILTER` statements), and support for "multi-edges" within Visual Query Graphs. For instance, multi-edges could enable users to specify multiple valid properties between items, simplifying the querying of ambiguous relationships (e. g., `wdt:P802` "student", `wdt:P1066` "student of", and `wdt:P69` "educated at"). While tools like [1] support this functionality through `FILTER` and `REGEX` statements, the use of these statements are less intuitive, just like reified structures require much technical understanding.

Another planned enhancement is the inclusion of graph-execution results visualisation directly within the tool. Users would execute queries and display the results as interactive graphs, further streamlining the query process.

The current implementation supports only string literals. There are plans to introduce value-restricted fields in the User Interface for all XML Schema data types. Additionally, other Wikibase-specific features such as label inclusion in queries remain areas for future development.

Feature Description	Implementation Status
Drawing a VQG with variables and literals	✓
Searching for entities on multiple Wikibase instances	✓
Creating SPARQL-SELECT queries from a VQG	✓
Code editor for SPARQL queries	✓
Applying changes in the code editor to the VQG	(✓)
Enriching unseen entities with metadata from the Wikibase API	(✓)
Literals with standard RDF data types (string, int, date, …)	(✓)
Use multiple Wikibase instances as data sources	✓
Meta-Info Panel	✓
Rendering qualifiers with the proposed visualisation	×
Value Constraints	×
Result Modifiers (e. g. `ORDER`, `LIMIT`)	×

Table 2: An overview of all features currently implemented comparing with other approaches. "✓" means implemented and tested, "(✓)" means implemented but not bug-free and "×" means not implemented. A full feature list can be found in the technical documentation of the repository.

Bibliography

[1] H. Vargas, C. Buil-Aranda, A. Hogan, and C. López, 'RDF Explorer: A Visual SPARQL Query Builder', in *The Semantic Web – ISWC 2019*, C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois, and F. Gandon, Eds., Springer International Publishing, 2019, pp. 647–663.

[2] O. Simons, 'In einer Graphdatenbank müsste man eigentlich auch graphisch suchen können'. Accessed: Oct. 01, 2024. [Online]. Available: https://blog.factgrid.de/archives/2596°

[3] O. Hartig, P.-A. Champin, G. Kellogg, and A. Seaborne, 'RDF 1.1 Concepts and Abstract Syntax'. Accessed: Dec. 06, 2024. [Online]. Available: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/°

[4] O. Hartig, P.-A. Champin, G. Kellogg, and A. Seaborne, 'RDF 1.2 Concepts and Abstract Syntax'. Accessed: Dec. 06, 2024. [Online]. Available: https://www.w3.org/TR/2024/WD-rdf12-concepts-20241121/°

[5] W3C, 'W3C SPARQL Language Specification'. Accessed: Nov. 03, 2024. [Online]. Available: https://www.w3.org/TR/rdf-sparql-query/°

[6] 'Wikibase RDF Mapping Article'. Accessed: Nov. 15, 2024. [Online]. Available: https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format°

[7] J. Lehmann *et al.*, 'DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia', *Semantic Web*, vol. 6, pp. 167–195, 2015, [Online]. Available: https://api.semanticscholar.org/CorpusID:1181640°

[8] M. F. Schönitzer, *Wikibase RDF Mapping Graphic (CC BY 4.0; no changes)*. Accessed: Nov. 15, 2024. [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=63880194°

[9] W3C, 'W3C SPARQL Formal Definition'. Accessed: Dec. 01, 2024. [Online]. Available: https://www.w3.org/2001/sw/DataAccess/rq23/sparql-defns.html°

[10] Wikibooks contributors, 'SPARQL/WIKIDATA Qualifiers, References and Ranks'. Accessed: Nov. 01, 2024. [Online]. Available: https://en.wikibooks.org/wiki/SPARQL/WIKIDATA_Qualifiers,_References_and_Ranks°

[11] F. Erxleben, M. Günther, M. Krötzsch, J. Mendez, and D. Vrandečić, 'Introducing Wikidata to the Linked Data Web', in *The Semantic Web - ISWC 2014*, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, and C. Goble, Eds., Springer International Publishing, 2014, pp. 50–65.

[12] C. Yang, X. Wang, Q. Xu, and W. Li, 'SPARQLVis: An Interactive Visualization Tool for Knowledge Graphs', in *Web and Big Data*, Y. Cai, Y. Ishikawa, and J. Xu, Eds., Springer International Publishing, 2018, pp. 471–474.

[13] T. Francart, 'Sparnatural: A Visual Knowledge Graph Exploration Tool', in *The Semantic Web: ESWC 2023 Satellite Events*, C. Pesquita, H. Skaf-Molli, V. Efthymiou, S. Kirrane, A. Ngonga, D. Collarana, R. Cerqueira, M. Alam, C. Trojahn, and S. Hertling, Eds., Springer Nature Switzerland, 2023, pp. 11–15.

Abbreviations

W3C.....World Wide Web Consortium (registered trademark)
RDF.....Resource Description Framework
RDFS.....Resource Description Framework Schema (Ontology within RDF)
SPARQL.....SPARQL Protocol And RDF Query Language (see Section 3.1)
IRI.....Internationalised Resource Identifier (see Section 2.1.1)
BGP.....Basic Graph Pattern (Definition 3.1)
OWL.....Web Ontology Language
VQG.....Visual Query Graph (see Definition 4.2)
VQL.....Visual Query Language
WASM.....Web Assembly
API.....Application Programming Interface
WWW.....World Wide Web

Appendix

Index of Figures

Index of Tables

Index of Listings

Use of Generative AI

This bachelor thesis was written in assistance of the OpenAI large language models GPT-4o and GPT-o1 preview. The large language models were used to ease literature research and to point out stylistic, orthographical, grammatical mistakes and to make formulation suggestions to the writer.

6. Declaration of Academic Integrity

1. I hereby confirm that this work — or in case of group work, the contribution for which I am responsible and which I have clearly identified as such — is my own work and that I have not used any sources or resources other than those referenced.

   I take responsibility for the quality of this text and its content and have ensured that all information and arguments provided are substantiated with or supported by appropriate academic sources. I have clearly identified and fully referenced any material such as text passages, thoughts, concepts or graphics that I have directly or indirectly copied from the work of others or my own previous work. Except where stated otherwise by reference or acknowledgement, the work presented is my own in terms of copyright.

2. I understand that this declaration also applies to generative AI tools which cannot be cited (hereinafter referred to as "generative AI").

   I understand that the use of generative AI is not permitted unless the examiner has explicitly authorised its use (Declaration of Permitted Resources). Where the use of generative AI was permitted, I confirm that I have only used it as a resource and that this work is largely my own original work. I take full responsibility for any AI-generated content I included in my work.

   Where the use of generative AI was permitted to compose this work, I have acknowledged its use in a separate appendix. This appendix includes information about which AI tool was used or a detailed description of how it was used in accordance with the requirements specified in the examiner's Declaration of Permitted Resources. I have read and understood the requirements contained therein and any use of generative AI in this work has been acknowledged accordingly (e. g. type, purpose and scope as well as specific instructions on how to acknowledge its use).

3. I also confirm that this work has not been previously submitted in an identical or similar form to any other examination authority in Germany or abroad, and that it has not been previously published in German or any other language.

4. I am aware that any failure to observe the aforementioned points may lead to the imposition of penalties in accordance with the relevant examination regulations. In particular, this may include that my work will be classified as deception and marked as failed. Repeated or severe attempts to deceive may also lead to a temporary or permanent exclusion from further assessments in my degree programme.

Place and date	Signature