



Bachelor Thesis

Semantic SPARQL Templates

Christina Davril

Examiner: Prof. Dr. Hannah Bast

November 14th, 2023

Faculty of Engineering

Department of Computer Science

Chair of Algorithms and Data Structures

universität freiburg

Writing Period

16.08.2023 – 16.11.2023

Examiner

Prof. Dr. Hannah Bast

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids other than those listed have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg im Breisgau, 14.11.2023



Place, Date

Signature

Abstract

Question Answering over Knowledge Graphs (KGQA) remains a very challenging problem, with some seemingly straightforward questions requiring complex, nested queries to be answered. This thesis presents a novel, template-based approach by identifying knowledge graph-independent "semantic SPARQL templates". These templates constitute a comprehensive set of syntactic building blocks used to realize semantic aspects of users' information needs.

To assess the practical utility of the found templates, they were applied to a newly created benchmark. This benchmark, called *Wikipedia Lists*, leverages Wikipedia lists as a verifiable ground truth, ensuring, to some degree, the correctness of the query. The benchmark covers a wide range of query structures, addressing a limitation found in most other Wikidata-based benchmarks.

The "semantic SPARQL templates" have been applied successfully to all examples in the benchmark with the results of the template-based queries matching those of their hand-written counterparts.

Moreover, using the templates may improve the quality of the resulting queries compared to queries written by hand. The general applicability of the templates translates to robust queries that return the desired outputs with varying input data. This stands in contrast to human-curated queries, which may rely on world knowledge, which can be incorrect or may only be applicable during a certain time.

In addition, the application of the templates to the benchmark provided information about the templates' usage frequencies, shedding light on structural patterns that appear in SPARQL queries and how they relate to patterns in the natural language (NL) question.

In summary, this thesis establishes a foundation for a semantic templates-based KGQA system, while also contributing a new Wikidata-based benchmark and offering preliminary findings about structural patterns in Wikidata-based SPARQL queries.

Zusammenfassung

Question Answering over Knowledge Graphs (KGQA), die Beantwortung natürlichsprachlicher Fragen mittels Wissensgraphen, stellt nach wie vor ein herausforderndes Problem dar. In manchen Fällen werden zur Beantwortung simpel erscheinender Fragen komplexe, verschachtelte Querys (Abfragen) benötigt. Diese Arbeit stellt einen neuartigen Ansatz vor, bei dem sogenannte "semantische SPARQL-Templates" identifiziert werden. Diese Templates bestehen aus syntaktischen Bausteinen, die zur Realisierung semantischer Merkmale der Informationsbedürfnisse der Nutzer:innen gebraucht werden. Zusammen decken diese Wissensgraph-unabhängigen Templates die meisten Anfragen ab.

Die Templates wurden auf einen neu erstellten Benchmark angewandt, um ihren praktischen Nutzen zu bewerten. Dieser Benchmark namens *Wikipedia Lists* nutzt die Informationen in Wikipedia-Listen als eine überprüfbare "Ground Truth" und sichert damit bis zu einem gewissen Punkt die Korrektheit der Query. Der Benchmark deckt eine große Bandbreite von Query-Strukturen ab, was bei anderen Benchmarks häufig nicht gegeben ist.

Die semantischen SPARQL-Templates konnten erfolgreich auf alle Beispiele des Benchmarks angewandt werden. Die Ergebnisse der mit Templates generierten Querys stimmten mit denen entsprechender handgeschriebener Querys überein.






Die Qualität von Querys kann durch die Nutzung von Templates gegenüber handgeschriebenen Versionen potenziell verbessert werden. Grund dafür ist die allgemeine Anwendbarkeit der Templates und der damit generierten Querys. Bei der Erstellung handgemachter Querys wird hingegen ggf. Weltwissen miteinbezogen, welches falsch oder nur zeitlich begrenzt gültig sein. Des Weiteren lieferte die Anwendung der Templates auf den Benchmark Informationen über ihre Nutzungshäufigkeiten. Diese geben Aufschluss über in SPARQL-Querys vorkommende Muster und ihr Verhältnis zu Mustern in der natürlichsprachlichen Frage.

Zusammenfassend legt diese Arbeit den Grundstein für ein auf semantischen Templates basierendes KGQA-System, trägt einen weiteren auf Wikidata basierenden Benchmark bei und bietet erste Erkenntnisse über strukturelle Muster in auf Wikidata basierenden SPARQL-Querys.

Contents

1	Introduction	1
1.1	The Problem of KGQA	1
1.2	Translating Semantics to Syntax	3
2	Related Work	7
2.1	Approaches Using <i>Pre-Made</i> Templates for Structures in the KG	7
2.2	Approaches Using <i>Generated</i> Templates for Structures in the KG	8
2.3	Approaches Using <i>Pre-Made</i> Templates for Full SPARQL Queries	9
2.4	Approaches Using <i>Generated</i> Templates for Full SPARQL Queries	10
3	Wikidata-Based Benchmarks	13
3.1	Evaluation of Existing Benchmarks	13
3.2	New Wikidata-Based Benchmark: <i>Wikipedia Lists</i>	17
3.2.1	Process of Creation	18
3.2.2	Evaluation	20
4	Approach	23
4.1	Syntactic Considerations	23
4.2	Template Selection and Naming	24
4.3	Excluded SPARQL 1.1 Constructs	25
4.4	Excluded Query Types	26
4.5	The Problem of Missing Values	26
5	Semantic Categories and Templates	27
5.1	Semantic Categories	28

5.2	Semantic Templates	30
5.2.1	Triple Patterns to Retrieve Selected KG Data	31
5.2.2	Imposing Relations by Connecting Variables	33
5.2.3	Aggregates	34
5.2.4	Arguments of Aggregates	37
5.2.5	Ranked Values	39
5.2.6	Arguments of Ranked Values	40
5.2.7	Negative Characterizations	41
5.2.8	Characterization Alternatives	42
5.2.9	Filtering Values and Ensuring the Inequality of Variables	42
5.2.10	Definition of New Variables Using Expressions	43
5.2.11	Optional Attributes	43
5.2.12	Add Names	44
5.2.13	Add Descriptions	45
5.2.14	Project Output Variables	45
5.2.15	Order Output by Variables	46
5.3	Shortcomings of the Established Templates	46
5.3.1	Resource-Use of the "connect" Template	46
5.3.2	Arbitrary Containment Relations	47
5.3.3	Constraints and Expressions in FILTER and BIND as String Inputs	47
5.4	Further Observations Regarding the Established Templates	47
5.4.1	Non-Deterministic Usability of Templates	47
5.4.2	Hidden Frequency Differences of Aggregators	48
5.4.3	Possible Template Additions	48
6	Usage of Templates	49
6.1	Example Usage of Templates	49
6.1.1	Example 1: The city with most museums for each country.	50
6.1.2	Example 2: How many countries have never been a member of the UN?	53
6.1.3	Example 3: Which US president was played by the most actors in a movie?	54
6.1.4	Example 4: Famous twins	57
6.1.5	Example 5: List of countries whose capital is not their largest city	62

6.1.6	Example 6: What is the combined total revenue of the three largest Big Tech companies ordered by number of employees?	65
6.1.7	Example 7: How many years did the second oldest dog in the world live?	69
6.1.8	Example 8: NSAID compounds with molecular weight < 200 g/mol	71
6.2	Analysis of Semantic Alternatives	74
6.2.1	 ATTRIBUTE	74
6.2.2	 CONSTRAINT	74
6.2.3	 AGGREGATE	75
6.2.4	 COMBINE	75
6.2.5	 OUTPUT	75
6.2.6	Conclusions	75
6.3	Analysis of Template Usage	76
6.3.1	Template Counts	76
6.3.2	Template Containment Relations	78
6.3.3	Conclusions	81
7	Conclusion	83
8	Acknowledgements	85
9	Appendix	87
	Bibliography	92

List of Figures

1	Example of a question-query pair	5
2	Snippet of a Wikipedia list (left) and corresponding QLever output (right)	18
3	Template counts for <i>Wikipedia Lists</i>	76
4	Template containment relations with grouping for <i>Wikipedia Lists</i>	78
5	Syntactic structure of NL questions that may correspond to queries with <code>arg_agg_all</code> containing <code>agg</code>	80
6	Template containment relations without grouping for <i>Wikipedia Lists</i>	87

List of Tables

1	Wikidata-based benchmarks	14
2	Complexity of the <i>Wikipedia Lists</i> benchmark	20
3	Top-level structure (Example 1)	50
4	Partial structure (Example 1): [1] number of museums per city	51
5	Structure (Example 2)	53
6	Top-level structure (Example 3)	54
7	Partial structure (Example 3): [1] number of actors that played a US president in a movie per president, and the names of these actors	55
8	Partial structure (Example 3): [1.2] actors who played a US president in a movie	56
9	Top-level structure (Example 4)	57
10	Partial structure (Example 4): [1] twins with birthdate, and name and description of twin sibling if available	58
11	Partial structure (Example 4): [1.3], twins (characterized indirectly) with name and description of twin sibling and shared birthdate	59
12	Partial structure (Example 4): [1.3.1] date of birth per twin with precision 11 or higher	60
13	Partial structure (Example 4): [1.3.2] mother per twin	61
14	Top-level structure (Example 5)	62
15	Partial structure (Example 5): [1] countries whose capital is not their largest city	63
16	Top-level structure (Example 6)	65
17	Partial structure (Example 6): [1] most recent total revenue per Big Tech company with number of employees on ranks 1 through 3 (descending order)	66

18	Partial structure (Example 6): [1.1] Big Tech companies with number of employees on ranks 1 through 3 (descending order)	67
19	Structure (Example 7)	69
20	Top-level structure (Example 8)	71
21	Partial structure (Example 8): [1] NSAID compounds with molecular weight smaller than 200 g/mol. Show names of NSAIDs.	72
22	Partial structure (Example 8): [1.1] NSAIDs (non-steroidal anti-inflammatory drugs)	73

1 Introduction

1.1 The Problem of KGQA

While search engines like Google return web pages that its algorithm deems relevant for the user based on the search terms they entered, it can only ever return existing web pages created, one way or another, by a person.

If a user wants very specific information, that information may be spread across multiple web pages. The user would have to compile the desired results themselves by e.g., lots of copy-pasting, all the while risking mistakes.

In contrast, if the desired information is available on Wikidata, the user would likely be able to access this structured information and output it in the desired format using only a corresponding SPARQL query.

For example, one could query for painters who were born in Germany but died in France. It stands to reason that compiling this information by hand would be tedious. Given a suitable [query](#), Wikidata outputs the 1,000+ people matching the description in a fraction of a second. With a small addition to the query text, it can sort the results by the number of Wikipedia site links – a proxy for their popularity.

If we assume that the knowledge graph, represented in RDF (**R**esource **D**escription **F**ramework) triples, contains all the required information, the main problem with this approach is that the user must first learn to use the SPARQL (**S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage) query language. Even having mastered SPARQL and using a query engine with an auto-completion feature, the user has to write the query texts themselves. This can be a time-consuming task as query texts are sometimes surprisingly long and complex, even if the underlying request, the user's **information need** or **intent**, appears simple enough.

Question **A**nswering over **K**nowledge **G**raphs (KGQA) systems allow for users to input NL

(natural language) questions or prompts into systems that query the underlying knowledge graph, attempting to return the desired information.

While so-called "simple questions" – usually defined as those that can be answered by a query containing only 1-2 triples – can nowadays be answered quite well, KGQA systems still struggle with more complex requests.

The idea to use templates to tackle this problem is not a new one. Several researchers have either identified templates by hand or generated them (semi-)automatically from data. However, the consensus in the research community is that the problem behind KGQA is too complex to be solved using any kind of template.

For example, Höffner et al. who published a survey about KGQA in 2017, describe template-based approaches as limited to "simple query structures" [1].

Accordingly, many template-based systems only use templates for simple examples, matching certain linguistic patterns in the natural language (NL) question to them, or instantiating them exhaustively until a result is found.

However, mapping aspects of the semantic structure of the user's question to one or more SPARQL templates might be a very useful step in KGQA systems. One could, for instance, determine the semantic structure using a system with a GUI that allows users to specify the semantic requirements without requiring any knowledge of SPARQL.

Moreover, having determined the required templates, one could analyze how the templates are usually used and combined and how this relates to patterns in the NL question.

This thesis explores this approach by identifying a set of semantics-based SPARQL templates and showing their application to a newly created benchmark, called *Wikipedia Lists*. This benchmark, presented in Chapter 3 was created due to shortcomings of existing KGQA benchmarks, mainly with regard to the variety of query types.

1.2 Translating Semantics to Syntax

Humans who are well versed in using query languages like SPARQL, and who want to write a query – e.g., to retrieve all German politicians and their most recent party – to run over a knowledge graph, take the following steps:

1. Analyze the semantic structure of the question they want to answer using the KG
2. Translate that semantic structure into a SPARQL query
 - a) Determine the required syntactic structure
 - b) Determine elements to be put into the syntactic structure (e.g., IRIs of items or properties, or literals used as parameters)

In the example, a sketch of the semantic structure of the question – or more fittingly: request – could look like this:

- entities should be politicians
- entities should be German
- entities should be output with their most recent party

Looking at each characteristic separately, we might call the first and second ones "constraints". While being a politician might be seen as an attribute of a person, in the context of a knowledge graph, we reduce the set of all potential entities in the KG to a subset of entities that fulfill the constraint of being politicians.

Adding the constraint that the entities should be German, we exclude politicians who aren't German.

Adding the current party of each politician to the output is, however, not a constraint. Up to this point, our set of entities would contain the IRIs of German politicians in a single column. By adding the most recent party for each person, we gain information, and we would likely call this added information an "attribute", becoming manifest as an additional column of our output. This is an updated version of the sketch:

- entities should be politicians – CONSTRAINT
- entities should be German – CONSTRAINT
- entities should be output with their most recent party – ATTRIBUTE

Now that the rough semantic structure has been established, humans – just like KGQA systems – face the challenge of finding a suitable syntactic representation for it.

For this, we can tackle each characterization separately. Combining them is later solved by a shared variable for the entities.

If we want to realize the **CONSTRAINT that entities should be politicians**, we are likely to do so by adding a triple to the output that links our entity variable to a class "politician" by some single property or a property path containing multiple elements.

While there are more complex syntactic means to fulfill a constraint, like using the *filter* or *arg max* construct, these are not applicable here.

For the **CONSTRAINT that entities should be German**, the situation is the same.

The **ATTRIBUTE of each entity's most recent party** is a more interesting case. There are several syntactic possibilities to realize an ATTRIBUTE:

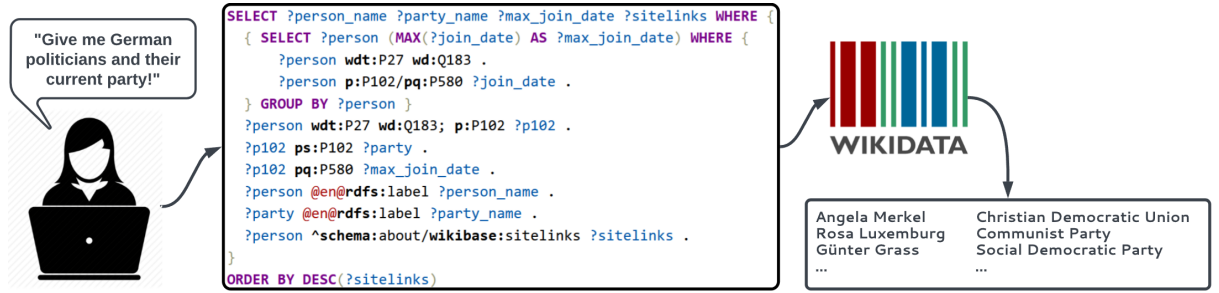
- If a property "most recent political party" exists, it can be used as the predicate of a single triple.
- We may need to perform an aggregation involving one or more variables. Possible aggregations are ...
 - ... a *count*, *sum*, *mean*, *max*, ... (overall or across formed groups)
 - ... an *arg max*, *arg min* ... (overall or across formed groups)
- We might need to combine information from different variables, defining the required variable ourselves.

If the KG we use is Wikidata, we do not have a property like "most recent political party", but instead "member of political party" statements, provided with a value (i.e., the IRI of a political party) and – among others – a qualifier "start time".

As humans, we know that this is the material we need to "define" a variable containing the "most recent political party" using the *arg max* aggregation: We want to retrieve the values of the "member of political party" statements with maximum "start time" qualifier values.

This section gave some insight into one reason why semantic SPARQL templates are useful: if we can categorize aspects of the semantic structure into categories like CONSTRAINT or

Figure 1: Example of a question-query pair



Source: <https://ad-blog.cs.uni-freiburg.de/post/semantic-sparql-templates-for-question-answering-over-wikidata>

ATTRIBUTE, this might narrow down the syntactic constructs that are needed to realize them. Which template is the right one can, among other things, be derived from cues in the NL question.

In the following, Chapter 2 will give an overview of template-based approaches to KGQA.

Existing Wikidata-based benchmarks for KGQA, as well as the new *Wikipedia Lists* benchmark will be presented in Chapter 3.

Chapter 4 will detail which types of queries are covered by the approach, which SPARQL constructs are used in the templates, as well as other methodological details.

The templates are then presented in Chapter 5, along with the other identified semantic categories, encompassing semantic purposes at a higher, more abstract level.

Chapter 6 will show how a more detailed semantic structure, called a **semantic plan**, can be translated unambiguously into a suitable SPARQL query, using **semantic templates**. This is done using a small set of eight examples. This set is also used to analyze how often the semantic templates realize the semantic categories (e.g., ATTRIBUTE, CONSTRAINT).

This section also includes an analysis of how often the semantic templates are used in their application to the *Wikipedia Lists* benchmark, how the templates are combined, and how some combinations relate to certain patterns in the NL languages.

Lastly, Chapter 7 concludes the thesis by summarizing its key findings and suggesting possible directions for future work.

2 Related Work

In the literature on KGQA, there are many template-based approaches. However, no approach aims to provide a complete set of semantics-based, basic SPARQL templates to serve as building blocks for the construction of full queries.

This section gives an overview of template-based approaches to KGQA published between 2012 and 2023. Höffner et al. write in their 2017 survey on KGQA that the development of template-based approaches seems to have decreased after 2013 [1]. Indeed, many articles using template-based approaches are relatively old.

Note that the word "question" is used for the user input to distinguish it from the word "query", reserved for the SPARQL text. "Question" should, however, also encompass keyword-style user input or input phrased as an imperative.

Due to the large number of publications, this section just lists a few examples that cover the main strategies and procedures used when tackling the problem of KGQA using templates. The approaches chosen are all open domain, monolingual, and either use a single KG or are KG-independent.

2.1 Approaches Using *Pre-Made* Templates for Structures in the KG

Lopez and Motta (2012) aim to extract so-called *linguistic triples* from the NL question [2]. These triples, which may be interlinked, are then mapped to *ontological triples*, representing subgraphs in the KG. Their system, *PowerAqua*, is often cited as a template-based system, and it can indeed be seen as using a single, pre-made template that represents the basic structure of a triple.

As a result, the system is only suitable for questions that are structurally very simple and do not

require subqueries, aggregation, or the operators **ORDER BY**, **LIMIT**, **OFFSET**, **MINUS**, **UNION**, **BIND**, or **FILTER**.

A very similar approach was presented by He et al. (2013) and applied in their system *Casia* [3]. Rahoman and Ichise (2013) present an approach in which they use a fixed set of templates to connect two KG resources. The resources can either be items (serving as subjects and objects of triples) or properties (serving as predicates of triples). The templates here span at most two triples [4].

In a relatively similar approach, Bast and Haussmann (2015) use three templates to represent graph structures connecting an entity mentioned in the NL question and the answer node [5]. In their system, *Aqqu*, the templates are instantiated exhaustively and cover structures which, when translated into a SPARQL query, span at most three triples.

As mentioned before, all these approaches are only able to handle questions that are answerable with simple SPARQL queries that only contain a few triples.

2.2 Approaches Using *Generated* Templates for Structures in the KG

Shekarpour et al. (2015) use their system *SINA* to generate "query graph templates", i.e., basic graph patterns representing subgraphs of the KG, for each input question. This is done by identifying and then connecting the required resources in the KG. The generated templates are then filled and used to answer the user's question [6].

Abujabal et al. (2017) use a similar approach but learn templates from questions and their answers in an offline step [7]. Doing this, they break down so-called "complex questions", i.e., questions with coordinating conjunctions and relative clauses, into smaller questions before they learn templates for them. Their system, *Quint*, then matches questions to potential templates. If the question is complex, it is broken down into sub-questions whose corresponding template candidates are instantiated and stitched together by intersecting the sets of answers they produce.

All these approaches that purely work with structures in the KG are unable to deal with queries containing operators like **ORDER BY**, **GROUP BY**, **FILTER**, ... that are independent of KG structures.

2.3 Approaches Using *Pre-Made* Templates for Full SPARQL Queries

Unger et al. parse the NL question and convert its extracted semantic representation to a SPARQL template [8]. The resulting templates contain operators like **COUNT**, **FILTER** and **HAVING**, and they have one template for getting the arg max or arg min using **ORDER BY** and **LIMIT 1**. The article does not go into much detail about which linguistic patterns (e.g., quantifiers, comparatives, superlatives) are translated to which semantic templates. It is unclear which queries can be tackled by the approach but it appears that the produced templates are stand-alone and not meant to be combined.

Park et al. (2014) map different question types to SPARQL templates representing full queries [9].

They distinguish, among other things, between Boolean questions (answered using **ASK** queries) and non-Boolean "simple questions". The latter are answered using a query containing only a basic graph pattern with one or more triples. Furthermore, they differentiate between three types of "aggregation function questions" which they map to a template using **COUNT**, a template using **ORDER BY**, **OFFSET 0**, and **LIMIT n**, and a template using **FILTER**.

The mapping is performed using pattern matching: e.g., questions containing "how many" are mapped to the *count* template, questions with superlatives to the *order by-offset-limit* template, and questions with comparatives to the *filter* template. Here, too, the range of questions that can be answered is limited.

A similar approach was presented by Berant and Liang (2014) who map questions to five pre-made SPARQL templates which they refer to as "logical forms" [10]. The mapping is done based on the syntactic structure of the question. The limited coverage of these templates can be illustrated by the only template featuring aggregation taking an overall count of items.

A recent approach by Formica et al. (2023) uses a set of 21 pre-defined templates that they created following a classification of questions into 21 types [11]. Judging from the revised literature, this is the largest number of pre-made templates reported so far. However, the templates listed there are not always basic, as they represent full queries. For example, they use a question class "[Quantitative] Count over Atleast" which represents questions like "How many films did at least 9 people do the dubbing for?". Semantically, this combines an aggregation using **COUNT** and one for filtering using either **FILTER** or **HAVING**, along with templates used for the triples in the query body. Besides, one could certainly combine this template with the

"[Quantitative] Count over Atmost" template which would use " \leq " instead of " \geq " in the filtering clause. Variants for "<" or ">" are not listed at all in the article.

While all these approaches cover varying numbers of query types, the questions that can be answered with them are limited. Since the templates represent *full* SPARQL queries, they are not intended to be combined, e.g., by nesting one into the other. This would, however, be needed to answer questions like "Who won the most Oscars after the year 2000?" where we would have to combine a *count*, *arg max* and *filter* template.

2.4 Approaches Using *Generated* Templates for Full SPARQL Queries

There also exist approaches that *generate* a large number of SPARQL templates that represent full queries. However, the resulting templates are not always purely syntactic but contain KG-specific entities.

For example, Zheng et al. (2015) mention a template for answering questions of the format "Which <_> graduated from <_>?" [12]. This template would, thus, contain the KG's predicate for 'graduated from'. For Wikidata, this would likely be **P69** ("educated at").

More recent approaches use learning to generate millions of such templates. For example, Cui et al.'s 2017 system, called *KBQA*, learned close to 30 million templates containing roughly 3,000 predicates [13].

Zheng et al.'s 2018 system, *TemplateQA*, generates "simple" or "binary" templates by which they understand templates which "[contain] just one fact" [14]. Training the system on Wikipedia, DBpedia and Freebase, they generated ~ 5 million templates mapping NL question patterns, created from declarative sentences in the Wikipedia, to predicates in each of the KGs. When a complex question, i.e., one referring to multiple facts, is input, it is decomposed into multiple smaller questions which are then mapped to simple templates whose results are combined. "who acted in Mission Impossible and Vanilla Sky?" is an example they give for a complex question, containing two facts that would be solved by intersecting the results from the queries used to retrieve the actors starring in both films. The examples from the paper indicate that this approach is limited to answering questions requiring basic graph patterns. Using a similar approach, Ding et al. (2019) solve the problem of generalizing the learned templates to unseen query structures by looking at *query substructures* instead of structures of full queries [15]. Their

system, *SubQG* learns these substructures and their predictors, and then maps NL questions to combinations of substructures to generate queries. These substructures, however, do not correspond to semantic building blocks but rather to frequently occurring chunks of syntax found in SPARQL queries. Besides, their coverage is still limited as they do not cover queries requiring operators like **UNION** or **GROUP BY**.

A KG-independent approach, containing purely syntactic templates is presented by Gomes et al. (2022) [16]. They use a Tree-LSTM and an attention mechanism to determine the most important semantic information contained in an NL question. Corresponding templates are learned using a modified version of the *LC-QuAD 2.0* dataset. Using this procedure, the researchers identified 29 SPARQL templates for Wikidata – each representing a full, but slotted SPARQL query.

The detected templates cover a wide range of queries but are limited in two senses:

Firstly, they are fixed, full queries instead of building blocks that could be nested and combined. Secondly, they lack many types of queries such as all types requiring an aggregation other than **COUNT**. A more diverse benchmark than *LC-QuAD 2.0* would be needed to identify other types. The topic of benchmarks will be discussed in the following chapter.

3 Wikidata-Based Benchmarks

3.1 Evaluation of Existing Benchmarks

Ever since Wikidata [17][18] became publicly available in 2012, and the Wikidata Query Service [19] was launched in 2015, several KGQA benchmarks have made use of its vast amount of triples, amounting to roughly 18 billion in the year 2023.

The following table gives an overview of some of them, ordered by the year in which they were published. The citations link to publications presenting the benchmarks.

The column "Complex queries?" lists types of queries contained in the benchmark that can be considered "complex":

- queries containing modifiers (e.g., **GROUP BY**, **FILTER**, or aggregators like **MIN** and **MAX**)
- queries containing subqueries
- queries with a minimum of three triples

Table 1: Wikidata-based benchmarks

Name	Year	Creation	Examples	Complex queries?
2017	SQWD [20]	Adaptation of "Simple Questions" dataset to Wikidata	21,399	none
2018	WebQSP-WD [21]	Adaptation of "WebQSP" dataset to Wikidata	3,913	no gold queries available
2018	CSQA/CQA [22]	Semi-automatically generated	1,600,000	no gold queries available
2019	LC-QuAD 2.0 [23]	Semi-automatically generated	30,000	queries with 3+ triples, COUNT and FILTER; no grouping, other aggregators or subqueries
2020	KQA Pro [24]	Automatically generated	117,970	queries with 3+ triples; no grouping, aggregators or subqueries
2022	QALD-9-plus [25]	Human-curated	412	3+ triples, grouping, aggregators, subqueries
2022	QALD-10 [26]	Human-curated	394	3+ triples, grouping, aggregators, subqueries
2022	WDBench [27]	Selected from real query logs [28]	2,658	queries with 3+ triples; no grouping, aggregators or subqueries

Among the listed benchmarks all but WDBench use synthetic datasets.

However, WDBench only considers real queries that contain basic graph patterns, including property paths, and **OPTIONAL** clauses. Other queries were pruned to contain only those elements. [27] Even though some of the datasets providing gold queries claim to contain "complex" examples, only the QALD-9-plus and QALD-10 datasets – the current most recent datasets in the QALD (Question Answering over Linked Data) series of KGQA benchmarks and competitions – contain *all* above-mentioned types of complex queries.

In a previous work [29], we identified 149 out of the 412 examples in QALD-9-plus (36%), and 220 out of the 394 examples in QALD-10 (56%), as complex.

This is in line with the benchmark authors' own analysis, according to which QALD-10 contains the modifiers **COUNT**, **FILTER**, **GROUP BY** and **OFFSET** significantly more often than previous benchmarks. [26]

Even though QALD-9-plus and QALD-10 present a stark improvement over older benchmarks when it comes to the spectrum of queries they cover, their overall quality could be improved in the following aspects:

- **SELECT DISTINCT** might be used consistently – or at the least *where needed* – in the outermost query in cases where duplicate results might occur in the output. For example, QALD-9-plus, ID 4, contains many duplicate results.
- **wdt:P31/wdt:P279*** might be used consistently – or at the least *where needed* – to deal with the structural inconsistency that is inherent in Wikidata already *by design*. For example, not all Q-items of cities in Wikidata are instances of the Q-item Q515, "city". Instead, many cities are instances of a subclass of "city" and are only retrieved when using **wdt:P31/wdt:P279*** instead of **wdt:P31** as is for example done in QALD-9-plus, ID 29.
- Some **gold queries do not match their NL question**, e.g., QALD-9-plus, ID 34, for which the gold query provides the count of items instead of the items themselves, or QALD-10, ID 46, with the inverse problem.

Further examples:

- QALD-9-plus, ID 44, asks about the current spouse of a person ("Who is the daughter of Robert Kennedy married to?") while the gold query also returns ex-spouses
- QALD-9-plus, ID 110, asks about rivers and lakes in South Carolina, but the query only considers lakes.
- QALD-9-plus, ID 241, asks about the smallest German city by area, yet the query uses the city's population values instead.
- Other examples have a **wrong set of answers**.
 - This can be because the **queries do not account for ties**. For example, QALD-9-plus, ID 143 asks "Who is the tallest player of the Atlanta Falcons?". If two players are tied for being the tallest, a random one among them will be output.
 - Another reason for this can be that the **queries don't use a required Wikidata entity**. For example, according to the benchmark, QALD-9-plus, ID 15, would output the entities "Victoria" and "George III of Great Britain". However, "George IV of the United Kingdom" and "William IV" should also be among the results for this

NL question: "Which monarchs of the United Kingdom were married to a German?". They are included when using the item **Q111722535**, "monarch of the United Kingdom of Great Britain and Ireland" referring to the period from 1801 to 1927. Looking at the query, it becomes apparent that only some periods of British history are included. Surprisingly, the item for "king" but not for "queen" is used.

- Another reason can be the **inadvertent use of world knowledge** – be it accurate or inaccurate. One example where this was done is QALD-9-plus, ID 165, which is about movies directed by Park Chan-wook. The query does not ensure that the result items are movies, leading to the inclusion of a TV mini-series the director – mostly known for his films – directed.

In QALD, ID 114, the query just uses "Mark Zuckerberg" (**Q36215**) as "the founder of Facebook" instead of querying this information.

Of course, these are just two examples of a broader category that might be labeled *human error*.

There are also the following, smaller problems concerning the usability of the benchmark:

- Some gold queries contain **syntactic flaws** like unused variables (e.g., QALD-9-plus, ID 85), missing **WHERE** operators, deviating from the standard syntax (e.g., QALD-9-plus, ID 30), or inconsistent capitalization of SPARQL operators (e.g., QALD-9-plus, IDs 407 and 408).
- Some gold queries contain **unnatural and/or unnecessarily complex syntax**. For instance, **FILTER(?a = ?b)** is used instead of simply employing a shared variable in QALD-9-plus, IDs 81 and 226.
- The **naming of variables** is very inconsistent and ranges from generic to descriptive, e.g., **?uri**, **?s1** (for "subject 1"), **?o1** (for "object 1"), **?c** (for "company"), **?bow** (for "body of water"), or **?islandgroup** and **?dateOfBirth**.

Even a perfect KGQA system will not always give the user the results they want due to the following problems:

- **ambiguous questions**, e.g., "Who was president of Croatia before milanovic?": The user may want *all* presidents coming before Milanović, as the preposition (indicating a range) suggests, or they may only be interested in the president immediately preceding Milanović.
- **vague questions**, e.g., "Which US American presidents were inaugurated at an old age?": What does the user consider to be an old age? Would an above-average age satisfy them?
- **implicit assumptions of the user**, e.g., asking a question like "Who was president in 1973?", a US-American user may refer to presidents of the USA. Similarly, asking about "countries", they might assume to only receive current and widely recognized countries.
- **missing or faulty data on Wikidata**

In general, one can also criticize making partially or fully human-curated benchmarks specifically for KGQA over Wikidata. If this procedure uses the information available on Wikidata as a starting point, it might nudge researchers towards creating examples that will be possible – and maybe even straightforward – to answer using Wikidata. While researchers can make a conscious effort to use paraphrased NL questions and synonyms to avoid using the exact labels of properties and items, and while they can intentionally look for harder examples, there *might* still be a subconscious bias to adjust the benchmark to the (current) state of Wikidata. This is of course problematic since users of KGQA systems would not present such a bias.




3.2 New Wikidata-Based Benchmark: *Wikipedia Lists*

To avoid the potential bias introduced by hand-curated benchmarks, created specifically for KGQA over Wikidata, and to include a verifiable ground truth, we developed the *Wikipedia Lists* benchmark. This benchmark is independent of Wikidata and uses the information stored in Wikipedia lists as ground truth. Wikipedia lists, often curated by experts or enthusiasts of the respective topic, contain information that people are interested in and might query a KGQA system about. This is especially true for Wikipedia lists with a lot of Wikipedia site links.

Figure 2: Snippet of a Wikipedia list (left) and corresponding QLever output (right)

Zone 5: South and Central Americas

- 500 –  Falkland Islands
 - 500 –  South Georgia and the South Sandwich Islands
- 501 –  Belize
- 502 –  Guatemala
- 503 –  El Salvador

?country_calling_code	?sample_flag_image	?country_name
+500		Falkland Islands
+500		South Georgia and the South Sandwich Islands
+501		Belize
+502		Guatemala
+503		El Salvador

Sources: en.wikipedia.org/wiki/List_of_country_calling_codes & qllever.cs.uni-freiburg.de/wikidata/fKDg2G

The benchmark, aptly called *Wikipedia Lists*, was created in March 2023 and contained 11 examples. It has since been expanded to contain **60 examples**. Most of these are **complete re-creations of tabular Wikipedia lists** or subsections thereof, e.g., of certain rows like the top 10 rows according to some ranking. In some cases, a **NL question derived from the information in a Wikipedia list** was used. For example, "Which American president was played by most actors in a movie? Also show the actors" is a question relating to the "List of actors who have played the president of the United States". We attempted to word these NL questions unambiguously and to provide all the necessary information regarding the desired query output.

Discrepancies between the Wikidata output and the Wikipedia list were resolved using other sources of information. While the data on Wikidata and in Wikipedia lists is sometimes incomplete or even faulty, the comparisons allowed to improve the quality of the queries, having them capture information that is sometimes stored structurally inconsistently on Wikidata.

Despite the Wikipedia lists being open source and possibly incomplete or faulty, we believe that providing and using a verifiable ground truth is necessary to develop scientifically and factually sound gold queries.

3.2.1 Process of Creation

The lists were chosen from various sources to cover a broad variety of topics and varying popularity levels of the lists, as measured by their Wikipedia site links:

- the Wikipedia "[List of lists of lists](#)" was browsed and a list was taken from each section
- Wikidata was queried for popular Wikipedia lists using the query shown below

- Wikidata was queried for Wikipedia lists with an ordering by ascending Q identifier, effectively yielding a random order

SPARQL 3.1: for Table 3

```
SELECT * WHERE {
  ?list wdt:P31 wd:Q13406463 . # instance of "Wikimedia list article"
  ?list @en@rdfs:label ?list_name .
  ?list ^schema:about/wikibase:sitelinks ?sitelinks .
}
ORDER BY DESC(?sitelinks)
```

The benchmark is stored as a **JSON file** using a reduced version of the format used in the QALD benchmarks. Each example contains the following information:

- example ID
- Boolean indicating whether aggregation is used in the queries
- NL question or indication that a full Wikipedia list was recreated
- link to the Wikipedia List
- QLever link to the hand-crafted query
- text of the hand-crafted query
- QLever link to the template-based query
- text of the template-based query
- section commenting on how well the Wikidata outputs and Wikipedia List information match

The semantic plans used to generate the template-based queries are not part of the benchmark as they are very long.

Due to the length of all involved files (e.g., the Wikipedia lists benchmark file, but also a file containing the semantic plans and various Python scripts to generate and analyze the template-based queries), they were not included in this written thesis.

While the results of the hand-crafted and generated queries largely matched, there were a few discrepancies due to QLever using varying orders for the elements in **GROUP_CONCAT** clauses. These discrepancies are resolved in the README.md file.

3.2.2 Evaluation

Some examples turned out to be problematic. For instance, the example with ID 9 fails to re-create the underlying Wikipedia list due to a lack of data. Instead, it tries to approximate it using the available data with moderate success. Concretely, since there is no information about drug sales on Wikidata, the query uses the popularity of drugs as indicated by their Wikipedia site links.

Other examples, like ID 17, make use of Wikidata-external knowledge. In ID 17, the conversion rates between different currencies on December 31, 2022, are needed.

Non-changing information, like factors needed to convert between SI units and non-SI units, were however taken from within Wikidata.

In IDs 3 and 33, due to the lack of a **NOW** function (as of November 10th, 2023), hardcoded values were used.

In general, many queries can be considered imperfect. There is no saying how they would perform with different inputs, i.e., new data in Wikidata, as they are adjusted to the current state of the Wikipedia lists and of Wikidata. Regarding this problem, the generated queries are likely more robust to changes in the input data. However, it should be pointed out that some of the handwritten queries were also – in part – made using the templates presented in Chapter 5, as the templates proved to be a useful aid during the construction of complex queries.

To show that the benchmark covers a variety of the more complex query structures listed at the beginning of this chapter, we indicate for each structure up to five examples (IDs) containing them in their handmade query versions. We also show the total number (absolute and relative) of examples whose handmade query version contains them.

Table 2: Complexity of the *Wikipedia Lists* benchmark

structure	example IDs	#	%
3+ triples	all IDs except 52	59	98
grouping	1, 2, 3, 4, 5	42	76
aggregation	1, 2, 3, 4, 5	47	78
subqueries	2, 4, 6, 7, 11	20	33

While only one-third of the examples contain a subquery, it should be noted that among these 20 examples containing them, 8 of the examples contain between two and four subqueries, while one example (ID 11, 'List of countries whose capital is not their largest city') contains a total of 21 subqueries. The high complexity of this query is due to having used templates even in the handwritten version, and due to the nesting of the *arg_agg* template which, by itself, already contains two subqueries.

This complex structure is needed to ensure correctness and to avoid duplicates in the output without including world knowledge. In this particular example, one can assume that it is relatively unlikely for a country to have two cities with the same most recent population count (i.e., a tie for being the largest city in the country), but this eventuality is considered by the query.

As a final note, it should be pointed out that in its current state, the benchmark is not ready to be used for training and testing KGQA systems.

For the examples with full list recreation, instead of providing links to the lists, one would have to provide detailed information about the desired columns, including the background knowledge contained in the description texts for the lists, as well as implicit information that can only be derived when analyzing the table contents.

For example, the Wikipedia "List of heads of state and government Nobel laureates" contains the years in which the heads of state and government won a Nobel Prize. The sub-section "In office" contains heads of state and government who won the Nobel Prize while they were in office. This sub-list uses this imprecise year information instead of the precise dates on which the people were awarded the Nobel prizes to determine whether the award was won "in office" or not. This only becomes apparent after closer inspection (cf. *Wikipedia Lists*, ID 16).

4 Approach

4.1 Syntactic Considerations

The precise syntax of the hand-written queries arises from the following considerations:

- **SELECT DISTINCT** was used consistently in the outermost query, even if it was not required, to eliminate duplicates in the output.
- **wdt:P31/wdt:P279*** was used consistently when retrieving instances of a certain class.
- The queries were made as natural and simple as possible, avoiding structures like **FILTER(?var1 = ?var2)**.
- The **names of the variables** were chosen to be informative, indicating which Wikidata entities they hold, and/or being named like the corresponding Wikipedia list column headers. Abbreviations were kept to a minimum to improve the usability of the benchmark.
- For small lists, attributes about entities – i.e., characteristics that don't define the set of result entities, but rather give additional information about them – were made **OPTIONAL** to include entities despite them lacking certain statements. This was done to make the Wikidata output similar to the Wikipedia list, allowing comparisons and ensuring the validity and completeness of the query.
- **UNION, VALUES** and **wdt:P279*** ("subclass of") have been used to **deal with structural inconsistencies** – both "inconsistencies" *by design* (e.g., items being instances of a class but not of superclasses of that class) as well as undesirable inconsistencies (e.g., use of different properties to store the same kind of information)

- Since **QLever** [30] was used to run the queries, only keywords and functions that are currently (October 2023) supported were used.

The syntax of the template-based queries is determined by the syntax of the templates. The following sections go into detail about how the templates were named and which SPARQL constructs and query types they cover. Notably, the **UNION** construct will be used in place of **VALUES** to deal with inconsistencies.

4.2 Template Selection and Naming

Chapter 5 will present the semantic SPARQL templates that have been identified. They contain placeholders for variables, IRIs and literals, making them knowledge graph-independent.

Some of the template names are equal to the names of SPARQL operators fulfilling the semantic aspect in question. While the semantic templates do not have a 1:1 correspondence to SPARQL operators, some of them naturally realize one particular, basic semantic aspect. For example, the template to project variables, *select*, uses the SPARQL **SELECT** operator. We chose to call it *select* rather than, say, *project*, as the former name immediately brings the template to mind.

The SPARQL 1.1 Query Language [31] specification explains which semantic effects can be achieved by using the various syntactic constructs (e.g., **FILTER** being usable to restrict numeric values). Apart from this, the specification uses a syntax-based categorization of its constructs. For example, the **LIMIT** and **OFFSET** modifiers are considered to be *solution sequence modifiers*, just like **SELECT**, **DISTINCT** and **ORDER BY**, as they are syntactic tools used to affect the output of a (sub-)query, called the *solution sequence*.

From a semantic point of view, these modifiers are usually used in certain combinations to achieve specific semantic effects. For example, to get the top three values of an ordered attribute, we use the *value_ranks_all* template (Section 5.2.5) that contains the modifiers **SELECT**, **ORDER BY**, **LIMIT** (with parameter 3) and **OFFSET** (with parameter 0) modifiers.

Similarly, the documentation lists the various aggregates – again, as syntactic tools – but does not contain any section about how to fulfill the semantic purpose that e.g., the *arg_agg* (Section 5.2.4) or *val_ranks_all* (Section 5.2.5) templates fulfill.

While the semantic meaning of each aggregate (e.g., taking a maximum or calculating a sum) is

different, their templates can be subsumed under general aggregation templates as they use the same syntax.

4.3 Excluded SPARQL 1.1 Constructs

The templates exclusively use SPARQL 1.1 and provide the required functionality to answer most user questions. The following SPARQL 1.1 Constructs are, however, syntactic sugar and/or replaceable by other constructs, such that they are excluded.

- blank nodes in basic graph pattern, replaceable by variables
- *predicate-object lists* as a shorthand notation for triple patterns with a common subject
- *object lists* as a shorthand notation for triple patterns with a common subject and predicate
- RDF collections
- the keyword **a** as alternative for **rdf:type**
- the **VALUES** construct, replaceable using **UNION**
- the **IN** and **NOT IN** functions, replaceable using **!=**, **=**, **&&** and **||**.
- the **HAVING** construct, replaceable using **FILTER**

It should be noted that some of these constructs, especially **VALUES**, **IN** and **NOT IN**, are very valuable in practical applications. They are simply excluded because they can be replaced by other, more widely applicable constructs. This is done to keep the number of semantic templates small and to list exactly one way to syntactically realize every semantic purpose.

Furthermore, the templates contain prefixed names as relative IRIs. The prefix declarations containing the base IRIs required to build the full IRIs are left out. Query engines like WDQS or QLever [30][32] often add them automatically.

While the commonly used constructs **FILTER EXISTS** and **FILTER NOT EXISTS** are not syntactic sugar, they can in almost all cases be replaced by other constructs and are also excluded: Instead of **FILTER EXISTS**, we can very often use a **VALUES** clause, and instead of **FILTER NOT EXISTS**, we can very often use **MINUS**.

For variables, **?** is consistently used instead of the alternative symbol **\$**.

4.4 Excluded Query Types

- **CONSTRUCT** queries
- **ASK** queries
- **DESCRIBE** queries
- federated queries
- queries containing **REDUCED**
- queries containing **GRAPH** or **FROM NAMED**

The first five query types are not required for querying a single, existent KG and providing the answers to the user's question as a set of distinct tuples.

Queries containing **GRAPH** or **FROM NAMED** are not included as these concepts are not supported by all KGs.

4.5 A Note about Missing Values

Even if KGs have ways to discern between missing values and meaningful absences of values (e.g., using the **wikibase:rank** property in Wikidata), we have to assume that this information is not always registered. For example, a person without any "spouse" statements in Wikidata may either never have been married, or they may have been but this information is not available in Wikidata. For this reason, we might get lower counts for the values of a statement – including more counts of zero – than we would if the KG was perfectly stocked. This also affects aggregates of count values.

Since there can be multiple reasons for the absence of statements, the way we treated missing values depended on the individual examples.

5 Semantic Categories and Templates

In this chapter, we will use four levels of abstraction (with level 1 being the most abstract and level 4 being the most concrete).

1. On the highest level of abstraction, we establish five **semantic categories** into which we can classify the semantic templates (Section 5.1). These categories are *independent of query languages and KGs*.
2. The **semantic templates' identifiers** – including the required parameters – represent lower-level semantic purposes. They are also *independent of query languages and KGs*.
3. The **semantic templates' syntax** represents the realization of the template in SPARQL. They use a fixed query language but are *independent of KGs*.
4. An **instantiation of a semantic template** with slotted-in arguments is a concrete query or a part of a concrete query. It uses a fixed query language and a fixed KG.

For example, if we want non-steroidal anti-inflammatory drugs (NSAIDs) in our output, this is the **semantic category** CONSTRAINT. This is because we reduce the set of all entities of a KG to the specified ones. Within this category, we then have multiple **semantic templates** that might be used to realize the CONSTRAINT. Among those, we might have the template with identifier **path** using a triple to realize a constraint. It requires the following parameters: a variable or IRI as the subject of the triple (**[varIRI]**), a predicate corresponding to the property **path** (**[pred]**), and an object that may be a variable, IRI or literal (**[obj]**).

One could now find a syntactic realization for this template in any query language that uses the RDF data model, e.g., RDQL or SeRQL. However, in this thesis, we use the syntactic realization using the **SPARQL query language** as it is the most widely used one.

This syntactic realization contains slots for the parameters that have to be filled with concrete entities from the KG that is used, as well as with concrete variables and literals. For example,

one might slot in the variable `?nsaid` for `[varIRI]`, the property path `wdt:P31/wdt:P279*` for `[pred]` and the entity `wd:Q188724` for `[obj]` if one uses **Wikidata as the underlying KG**. The only output of this query – realizing this CONSTRAINT – is currently Voltaren gel with the active ingredient diclofenac, an NSAID.

If we however use **PubChem as the underlying KG**, we get many more results (~24,000) at the cost of the above CONSTRAINT requiring more than a single triple to be realized. Instead, the CONSTRAINT is realized using one instance of `path` and three instances of `add_path` that each realize a sub-constraint. This is because we first need to define the class of NSAIDs as a restriction class concerning the property "has role", and fixate that entities must have at least one value of "non-steroidal anti-inflammatory drug" for this property. Only then can we retrieve instances of this restriction class by using a single triple, much like in the Wikidata case.

5.1 Semantic Categories

The semantic templates identified in Chapter 5 can fulfill one or more of the following, high-level semantic purposes, represented as categories. For most of these five categories, the intended meaning cannot be defined precisely. As such, the short descriptions for each of them should only serve to give a general idea of the category's meaning by providing an example semantic template. In the following, the term "items" is short for "entities and/or information about them" - referring to all possible (output) tuples. The pictograms used to represent each category will be explained later in this section.

1. ATTRIBUTE

Description: templates that *aim to* add information about items

Example: template `add_path` when introducing a new variable, e.g., the second triple in `?person wdt:P31 wd:Q5 . ?person wdt:P27 ?country_of_citizenship .`, adding citizenship information to the entities.

2. CONSTRAINT

Description: templates that *aim to* filter out certain items

Example: template `filter` to add a constraint on the values of an attribute using **FILTER**

Note that aggregates are considered to be a separate category, even if they filter out items with certain values (e.g., non-maximal values).

3. AGGREGATE

Description: templates that apply expressions over groups of solutions, returning exactly one value per group; the word "group" can refer to explicit groups, formed using **GROUP BY**, or to the overall group containing all items

Example: template `agg_all` with aggregation type argument **MAX** to retrieve the maximum value of an attribute for the overall group

4. COMBINE

Description:

- templates which define new variables from – usually multiple – existing variables

Example: template `bind` to assign the result(s) of an expression to a new variable using **BIND**

- templates which combine different characterizations into a single one

Example: template `union` to unite multiple characterizations of the same variable using **UNION**

5. OUTPUT

Description: templates which do not *aim to* change the set of output items but project the results to certain columns or determine the ordering of result items

Example: `select` and `order` are templates used for this exact purpose

The magnifying glass expresses that we need to access the KG for these categories. Categories without it are KG-independent. For COMBINE, a star was chosen to symbolize novelty. The star is made of two halves to indicate that we usually combine two or more things – variables or pieces of syntax. The eye was chosen for the OUTPUT category as the templates in this category determine what the user sees in the final query.

This categorization does not partition the set of templates as the same template can fall into multiple categories. For instance, `add_path` can realize an **ATTRIBUTE** or **CONSTRAINT**.

The reason why certain categories' meaning is based on what they *aim to* achieve, is that we cannot base the meaning on the actual effects that semantic templates in this category have on

the set of result items.

To illustrate this, when `add_path` is used to retrieve an `ATTRIBUTE`, we might end up filtering out some items if they don't have a value for the property in question.

The intended additive nature of `ATTRIBUTE` becomes however apparent when we consider that we always add a column to the output. Regarding `CONSTRAINT`, the aim is always to eliminate the items that don't fulfill the constraint, potentially reducing the set of items by filtering out certain rows.

5.2 Semantic Templates

This section contains both the **identifiers** and the **SPARQL syntax required** for each template. It was made sure that the templates were as general as possible. For example, for the aggregation templates (`agg`, `agg_all`) using `COUNT`, we always include counts of zero, as these may be needed for queries that require taking the mean of the counts, and as they may simply be a desirable part of the output. In real queries or queries found in benchmarks, zero counts are often omitted when only a maximum count which is expected to be greater than 0 should be output. However, since zero *could be* the maximum, these queries rely on world knowledge.

In cases where only counts of at least 1 should be included, this can be achieved by applying the `filter` template afterward. Similarly, if there is an abundance of zero counts in the results, these can be moved to the end of the results list by applying a correspondent ordering.

Whenever pieces of syntax are inserted into templates – which will be the case for the elements filling the slots `[cont]` and `[char[varX]]` – the system must check if the inserted piece of syntax is a subquery. In this case, it needs to be ensured that they are wrapped in curly braces.

For some templates, there can be an arbitrary number of arguments of a certain type. This is indicated by using a tuple notation.

In general, the templates were crafted such that they cover relevant practical examples, avoiding unnecessary complexity.

5.2.1 Triple Patterns to Retrieve Selected KG Data

Most SPARQL queries contain at least one triple pattern, also called a *basic graph pattern*. It is an RDF triple that usually contains at least one variable. For these patterns, there are two templates, depending on whether the triple pattern is used to establish a *base block*, i.e., a new query graph, or if it adds to an existing query graph.

Note that properties in the triple's predicate may be inverted (constituting an inverted path) using the caret symbol ($\hat{}$). Thus, if an attribute needs to be added to a set of entities stored in a variable, and this attribute is retrievable with the variable as the *object* rather than the subject of the triple, this is also captured by the template.

While, technically, a predicate consisting of a single property is called a property path, we only use the term *property path* for multi-element versions to avoid the more verbose term *multi-element property path*.

Property paths are used when it is sufficient to have implicit variables, i.e., the variables connecting two path elements need not be accessed or output.

The path elements are connected by a forward slash as a *sequence path*. They may also contain path modifiers like $*$ (zero-or-more path) or $+$ (one-or-more path) to indicate the number of times a certain path element may be added, e.g., `[prop1]/[prop2]([prop3])*`

More information about property paths can be found in the SPARQL 1.1. Query Language documentation [31].

For these templates, `[pred]` contains the full property path, connecting the subject and the object of the triple, with all slashes and path modifiers.

Path

Template identifier: path

Parameters:

- `[varIRI]`: the *semantic* subject of the triple: variable or IRI
- `[pred]`: single property or property path
- `[obj]`: the *semantic* object of the triple: variable, IRI or literal

Template 5.1: path

`[varIRI] [pred] [obj] .`

The terms *semantic* subject and *semantic* object allude to the following two ways this template is usually used in practice:

- The syntactic subject of the triple is a variable that is defined by the predicate, and by the object of the triple which is an IRI or literal.

Example: `?human wdt:P31 wd:Q5 .`, used to retrieve all instances of the class human, realizing a CONSTRAINT. The *semantic* subject – corresponding to the syntactic subject – is then the variable `?human` that is being defined.

- The syntactic subject of the triple is an IRI for which a variable containing new information is added using the predicate. However, this still constitutes a CONSTRAINT as we reduce the set of all entities in the KG to those stored in the variable.

Example: `wd:Q42 wdt:P18 ?image .`

As stated before, the predicate of the triple may be an inverted property path. Thus, in instantiations of this template, the positions of the *syntactic* subject and object of the triple may be swapped.

Example: `wd:Q5 ^wdt:P31 ?human .`

There is also a more rarely found case where we realize a CONSTRAINT using two variables which must both be considered to be the semantic subject. For example, `?spouse_1 wdt:P26 ?spouse_2 .` retrieves all pairs of married people, defining two variables at the same time.

Add Path

Realizes a CONSTRAINT or ATTRIBUTE using a single triple.

Template identifier: `add_path`

Parameters:

- `[varIRI]`: the *semantic* subject of the triple: variable or IRI
- `[pred]`: single property or property path
- `[obj]`: the *semantic* object of the triple: literal, IRI or variable
- `[cont]`: query graph to add to

Template 5.2: add_path

```
[cont]  
[varIRI] [pred] [obj] .
```

If a CONSTRAINT is realized, the *semantic* subject is a variable. The variable is further defined by this added triple. Example: `?person wdt:P27 wd:Q183` . ensuring that persons have the German citizenship.

If an ATTRIBUTE is realized, the *semantic* object is a variable – containing the new information. Example: `?person wdt:P27 ?country_of_citizenship` . outputting persons’ countries of citizenship.

Technically, one would, thus, need two versions of this template – one for each semantic purpose. However, since they share the same, very basic syntax, they are grouped.

Another reason for doing this is that ATTRIBUTES are sometimes just realized to realize other ATTRIBUTES or CONSTRAINTs. A common example of this is the access of statement nodes (e.g., using `p` in Wikidata) which often just serve to connect statement values and qualifier values. The nodes themselves, when output, do not carry any meaning. The semantic purpose of accessing statement nodes is, thus, not quite one of an ATTRIBUTE.

5.2.2 Imposing Relations by Connecting Variables

This template has the specific purpose of connecting two existing variables. Applying this template, the variables are required to have a certain relation to each other, as specified by the predicate of the connecting triple.

Template identifier: connect

Parameters:

- `[char[var1]]`: characterization of first variable to be connected
- `[char[var2]]`: characterization of second variable to be connected
- `[var1]`: subject of connecting triple: variable
- `[pred]`: single property or property path
- `[var2]`: object of connecting triple: variable

Template 5.3: connect

```
[char[var1]]  
[char[var2]]  
[var1] [pred] [var2] .
```

5.2.3 Aggregates

Aggregate Values of Attributes – Overall

Template identifier: agg_all

Parameters:

- ([distinct1], \dots): aggregation type modifiers $\in \{ \text{DISTINCT}, \emptyset \}$
- ([agg1], \dots): aggregation types $\in \{ \text{COUNT}, \text{MAX}, \text{MIN}, \text{AVG}, \text{SUM}, \text{GROUP_CONCAT}, \text{SAMPLE} \}$
- ([var1.1], \dots): variables whose values are aggregated
- ([var2.1], \dots): variables to store the aggregated values
- [cont]: query graph to add to

Template 5.4: agg_all, showing two aggregations

```
SELECT ( [agg1] ( [distinct1] [var1.1] ) AS [var2.1] )  
      ( [agg2] ( [distinct2] [var1.2] ) AS [var2.2] )  
WHERE {  
    [cont]  
}
```

This template covers aggregations of variables across the whole group, i.e., across all items rather than across specifically created groups.

If we want to perform **multiple aggregations**, we have an expression in the **SELECT** clause for each position of the first four arguments. The expressions are separated by spaces as indicated in the template above.

For **GROUP_CONCAT**, one can specify a (default) separator, e.g., a semicolon, to avoid separating the aggregated items by spaces. However, one could also add this symbol as another parameter of the template. ¹

¹For the template-based *Wikipedia Lists* queries, a semicolon was used as a separator throughout.

Template 5.5: agg_all, with one **GROUP_CONCAT** aggregation

```
SELECT ( GROUP_CONCAT ( [distinct1] [var1.1]; separator="; " ) AS [var2.1] )
WHERE {
    [cont]
}
```

Note that instead of the name of the aggregated variable, being set through arguments, one could also let the system determine appropriate names, e.g., (**COUNT**(?museum) AS ?num_museums)

Aggregate Values of Attributes – Across Groups

Template identifier: agg

Parameters:

- ([distinct1], \dots): aggregation type modifiers $\in \{ \text{DISTINCT}, \emptyset \}$
- ([agg1], \dots): aggregation types $\in \{ \text{COUNT}, \text{MAX}, \text{MIN}, \text{AVG}, \text{SUM}, \text{GROUP_CONCAT}, \text{SAMPLE} \}$
- ([var1.1], \dots): variables whose values are aggregated
- ([var2.1], \dots): variables to store the aggregated values
- ([var3.1], \dots): variables to group by
- [cont]: query graph to add to (case 'no COUNT aggregation')
- [cont1]: query graph characterizing the counted variable (case 'COUNT aggregation')
- [cont2]: query graph characterizing the variable(s) used for grouping (e.g., for which counts are retrieved) and of the variables used exclusively in non-COUNT aggregations (case 'COUNT aggregation')

If we have **multiple aggregations**, they are handled as described in Section 5.2.3. The case of **GROUP_CONCAT** is also handled just as in Section 5.2.3.

The parameter ([var3.1], \dots) is realized syntactically by separating the elements of the tuple by spaces.

Case: no COUNT aggregation

If the aggregation type is not **COUNT**, [cont] is used in the **WHERE** body.

Template 5.6: agg, showing two non-COUNT aggregations

```
SELECT ( [agg1] ( [distinct1] [var1.1] ) AS [var2.1] )
      ( [agg2] ( [distinct2] [var1.2] ) AS [var2.2] )
      ([var3.1], \dots)
WHERE {
    [cont]
}
GROUP BY ([var3.1], \dots)
```

Case: COUNT aggregation

If one of the aggregation types is **COUNT**, **[cont1]** is used and realized within an **OPTIONAL** clause, while **[cont2]** outside of it. The characterization of the counted variable is put into an **OPTIONAL** clause to include values of zero.

It is important that the variable(s) *for which* counts are retrieved are characterized outside of the **OPTIONAL** clause, and that the variable whose values are counted is characterized within it.

Template 5.7: agg, with **COUNT**

```
SELECT (COUNT ( [distinct1] [var1.1] ) AS [var2.1] ) ([var3.1], \dots)
WHERE {
    [cont2]
    OPTIONAL { [cont1] }
}
GROUP BY ([var3.1], \dots)
```

About the order of elements in the WHERE clause

It should be emphasized that the order of the **OPTIONAL** clause and the other section(s) in the **WHERE** clause must not be changed as this would affect the result. If the wrong order is used, we end up excluding counts of zero unless all counts would be zero. See this [website](#) for more information.

Combinations of COUNT and other aggregates: The combination of **COUNT** and **GROUP_CONCAT** occurs often in practice and is shown here to provide another example:

Template 5.8: agg, with **COUNT** and **GROUP_CONCAT**

```
SELECT ( COUNT ( [distinct1] [var1.1] ) AS [var2.1] )
      ( GROUP_CONCAT ( [distinct2] [var1.2]; separator="; " ) AS [var2.2] )
      ([var3.1], \dots)
WHERE {
    [cont2]
    OPTIONAL { [cont1] }
}
GROUP BY ([var3.1], \dots)
```

It is possible to include `[char[varX.1]]` both inside and *additionally* outside of the **OPTIONAL** clause and this will in fact be useful when applying the templates in practice.

Note that the current version of this template is designed to handle at most one **COUNT** aggregation, as this appears to be sufficient in praxis. To handle multiple **COUNT** aggregations, one would add an **OPTIONAL** clause for each of them.

5.2.4 Arguments of Aggregates

Arguments of Aggregates – Overall

Template identifier: `arg_agg_all`

Parameters:

- `[distinct]`: aggregation type modifier $\in \{ \text{DISTINCT}, \emptyset \}$
- `[agg]`: aggregation type $\in \{ \text{MAX}, \text{MIN}, \text{AVG} \}$
- `[var1]`: variable whose value(s) corresponding to the aggregated value should be output (*arg*)
- `[var2]`: variable whose values are aggregated
- `[var3]`: variable to store the aggregated value
- `([var4.1], \dots)`: additional variables to project
- `[cont]`: query graph to add to

Template 5.9: `arg_agg_all`

```
SELECT [var1] [var3] ([var4.1], \dots) WHERE {
    [cont]
    BIND ( [agg] ( [distinct] [var2] ) AS [var3] )
    FILTER ( [var2] = [var3] )
}
```

In practice, `[var1]` will usually represent an entity (e.g., Wikidata Q-item), while `([var4.1], \dots)` will represent attributes for these entities (e.g., their label).

Wikipedia Lists IDs 2 and 39 are practical examples that use the "additional variables to project".

For this template, it was made sure that ties were included. A simpler template using **ORDER BY** and **LIMIT 1** is therefore not generally adequate and might return an arbitrary entity among multiple tied entities.

Arguments of Aggregates – Across Groups

Template identifier: `arg_agg`

This template applies *arg_agg* with specified groups.

Parameters:

- `[distinct]`: aggregation type modifier $\in \{ \text{DISTINCT}, \emptyset \}$
- `[agg]`: aggregation type $\in \{ \text{MAX}, \text{MIN}, \text{AVG} \}$
- `[var1]`: variable whose value(s) corresponding to the aggregated value should be output (*arg*)
- `[var2]`: variable whose values are aggregated
- `[var3]`: variable to store the aggregated value
- `([var4.1], \dots)`: variables to group by
- `([var5.1], \dots)`: additional variables to project
- `[cont]`: query graph to add to

Template 5.10: `arg_agg`

```
SELECT [var1] [var3] ([var4.1], \dots) ([var5.1], \dots)
WHERE {
  {
    SELECT ( [agg] ( [distinct] [var2] ) AS [var3] ) ([var4.1], \dots)
    WHERE { [cont] } GROUP BY ([var4.1], \dots)
  }
  {
    SELECT ( [var2] AS [var3] ) [var1] ([var4.1], \dots) ([var5.1], \dots)
    WHERE { [cont] }
  }
}
```

For this template, as in Section 5.2.4, it was made sure that ties are included.

Some practical examples using the "additional variables to project" are *Wikipedia Lists* IDs 6, 13, 48, and 50. They are used when the **agg** template is combined with the **arg_agg** template, and when statement qualifier information that is not aggregated should be output.

5.2.5 Ranked Values

Template identifier: `val_ranks_all`

Similar to **agg_all** but instead of retrieving one aggregated value, we retrieve values on potentially multiple, specified ranks of a given ordering. Note that the ordered values are not necessarily distinct. For example, if we want to retrieve the elevation values of the world's two highest mountains (cf. *Wikipedia Lists*, ID 59), we get *two* elevation values for Mount Everest if we use the mountain's country statement as a way to ensure that the mountain is located on Earth. This is because it is located both in China and Nepal. To ensure that the mountain entities (in `[var1]`) occur only once in the output, we can use aggregation before applying `val_ranks_all`.

Parameters:

- `[order]`: direction of the ordering $\in \{ \text{ASC}, \text{DESC} \}$
- `[var1]`: variable whose values are ranked
- `[limit]`: natural number for the **LIMIT** clause, fixing the number of ranks to consider
- `[offset]`: natural number for the **OFFSET** clause, fixing that ranks start at `[offset] + 1`
- `([var2.1], \dots)`: additional variables to project
- `[cont]`: query graph to add to

Template 5.11: `val_ranks_all`

```
SELECT [var1] ([var2.1], \dots)
WHERE {
    [cont]
}
ORDER BY [order] ( [var1] )
LIMIT [limit]
OFFSET [offset]
```

5.2.6 Arguments of Ranked Values

Template identifier: `arg_ranks_all`

Similar to `arg_agg_all`, but instead of using an aggregated value, we use values on specified ranks according to a given ordering.

Parameters:

- `[var1]`: the variable whose values correspond to the values on the indicated ranks (*arg*)
- `[order]`: direction of the ordering $\in \{ \text{ASC}, \text{DESC} \}$
- `[var2]`: variable whose values are ranked (*ranks*)
- `[limit]`: natural number for the **LIMIT** clause, fixing the number of ranks to consider
- `[offset]`: natural number for the **OFFSET** clause, fixing that ranks start at n+1
- `([var3.1], \dots)`: additional variables to project
- `[cont]`: query graph to add to

Template 5.12: `arg_ranks_all`

```
SELECT [var1] [var2] ([var3.1], \dots)
WHERE {
  {
    SELECT [var2]
    WHERE { [cont] } ORDER BY [order] ( [var2] )
                LIMIT [limit]
                OFFSET [offset]
  }
  {
    SELECT [var1] [var2] ([var3.1], \dots)
    WHERE { [cont] }
  }
}
```

Note that this template applies the specified ordering to the projected variables. In practice, this template will often be slotted into the `select` template to get distinct results, and then into the `order` template. A practical example with "additional variables to project" are *Wikipedia Lists* IDs 6 and 37. As these examples show, "additional variables to project" may be necessary when the `arg_ranks_all` template is combined with the `agg` template, and when qualifier information that is not used in the `arg_ranks_all` template (as either `[var1]` or `[var2]`) should be output.

ORDER BY \dots **LIMIT** \dots **OFFSET** may use non-distinct, duplicate values. This may or may not be desirable. Thus, same as for the `val_ranks_all` template, we may need to use aggregation in `[cont]`

to ensure that values of distinct entities are used.

It is assumed that we only order by one variable, as this appears to be sufficient in practice.

Arguments of Ranked Values and Ranked Values Across Groups

Surprisingly, it is not trivial to establish templates `arg_ranks` and `val_ranks` which apply `arg_ranks_all` and `val_ranks_all` to specified groups. While queries that calculate these can be written in practice, they are computationally expensive (e.g., variants using the cross product) or must be pre-generated, depending on the value for `[limit]` (e.g., a variant using a combination of `MAX`, `MINUS`, and `UNION`).

To realize this semantic purpose efficiently, clearly, and without the need for pre-generating, it might be the best solution to extend the functionality of SPARQL engines. For instance, one could adapt the `PARTITION BY` and `RANK` constructs from SQL to SPARQL. These constructs enumerate the items in each specified group after applying a desired ordering to them. With these ranks available, the task of realizing `arg_ranks_grp` and `val_ranks_grp` becomes simple.

5.2.7 Negative Characterizations

Template identifier: `minus`

To indicate constraints that a variable should *not* fulfill, the `MINUS` construct can be used. It can be seen as a way to combine a regular, "positive" characterization with a "negative" one. If multiple negative characterizations should be combined, the negative characterization can use the `union` template (Section 5.2.8), or the `minus` template can be used multiple times.

Parameters:

- `[cont]`: query graph to add to (containing a positive characterization of the variable)
- `[char[var]]`: negative characterization of the variable

Template 5.13: `minus`

```
[cont]
MINUS { [char[var]] }
```

5.2.8 Characterization Alternatives

Template identifier: union

To combine multiple characterizations of a variable, the **UNION** construct is used.

Parameters:

- `([char1[var]], [char2[var]], \dots)`: characterization alternatives of the variable
- `[cont]`: query graph to add to

Template 5.14: union, with two characterization alternatives

```
[cont]
{ [char1[var]] }
UNION
{ [char2[var]] }
```

Additional characterization alternatives are appended as **UNION** { `[charX[var]]` } in the **WHERE**-clause.

5.2.9 Filtering Values and Ensuring the Inequality of Variables

This template is mostly used to constrain ranges for the values of a variable. The values are often (alpha)numeric or (parts of) date literals. The template also often used to enforce that two variables are unequal.

Template identifier: filter

Parameters:

- `[constraint[var1, \dots]]`: constraint
- `[cont]`: query graph to add to

Template 5.15: filter

```
{ [cont] }
FILTER ( [constraint[var1, \dots]] )
```

The **FILTER** constraint may contain various functions.

The position of the **FILTER** clause in the **WHERE** group is arbitrary as it is always evaluated after the group in which it appears has been evaluated [31].

[cont] is always be included in brackets as leaving them out may cause memory errors in some engines.

5.2.10 Definition of New Variables Using Expressions

Template identifier: bind

In some cases, a new variable has to be defined from existing ones by an expression that is not an aggregation. An example of this is a variable for the concept of 'population density' for which some KGs may not have a ready-made property. In these cases, the variable can be defined as the quotient of the 'population' variable and the 'area' variable – if available. A common use of this is the extraction of the year information in a date literal using **YEAR**.

Parameters:

- [expression[var1, \dots]]: expression
- [varX]: newly defined variable
- [cont]: query graph to add to

Template 5.16: bind

```
[cont]
BIND ( ( [expression[var1, \dots]] ) AS [varX] )
```

For the **bind** template, it is important to realize that **BIND** ends the basic graph pattern preceding it [31]. This means that the characterization of the variables used in the **BIND** expression must come before **BIND**, as is ensured by the template.

5.2.11 Optional Attributes

Template identifier: optional

This template always realizes an **ATTRIBUTE**. The **OPTIONAL** clause encompasses information that should be included *if available*.

Parameters:

- **[cont1]**: query graph to add to (non-optional)
- **[cont2]**: optional query graph

Template 5.17: optional

```
[cont1]
OPTIONAL { [cont2] }
```

5.2.12 Add Names

Template identifier: add_name

This template is needed for most queries and adds the labels of the items stored in the indicated variables as separate columns. It can also be used to add the aliases of items. The template is able to add several labels at once. As with the aggregation templates, the tuple parameters are realized position by position, as the template demonstrates. When working with a specific KG, one could fix the predicate to the respective value, e.g., **rdfs:label** for Wikidata.

Parameters:

- **([var1.1], \dots)**: variables for which labels should be added
- **([var2.1], \dots)**: variables storing the labels
- **[pred]**: the predicate used to retrieve the labels
- **([lang1], \dots)**: the language(s) to use for the labels
- **[cont]**: query graph to add to

Template 5.18: add_name, naming var1 and var2

```
[cont]
[var1.1] [pred] [var2.1] . FILTER ( LANG[var2.1] = [lang1] )
[var1.2] [pred] [var2.2] . FILTER ( LANG[var2.2] = [lang2] )
```

5.2.13 Add Descriptions

Template identifier: `add_desc`

This template works exactly like `add_name` except that it adds descriptions. Syntactically, it is identical to the `add_name` template but has a different semantic purpose. When working with a specific KG, one could fix the predicate to the respective value, e.g., `schema:description` for Wikidata.

Parameters:

- `([var1.1], \dots)`: variables for which descriptions should be added
- `([var2.1], \dots)`: variables storing the descriptions
- `[pred]`: the predicate used to retrieve the description
- `([lang1], \dots)`: the language(s) to use for the descriptions
- `[cont]`: query graph to add to

Template 5.19: `add_desc`, retrieving descriptions of `var1` and `var2`

```
[cont]
[var1.1] [pred] [var2.1] . FILTER ( LANG[var2.1] = [lang1] )
[var1.2] [pred] [var2.2] . FILTER ( LANG[var2.2] = [lang2] )
```

In the case of KGs with only one (default) language for the labels, the languages(s) parameter is the empty tuple and the **FILTER**s are left out.

5.2.14 Project Output Variables

Template identifier: `select`

Parameters:

- `[distinct]`: projection type modifier $\in \{ \text{DISTINCT}, \emptyset \}$
- `([var1], \dots)`: variables that should be selected to be part of the final output
- `[cont]`: query graph to add to

Template 5.20: select

```
SELECT [distinct] ([var1], \dots) WHERE {  
    [cont]  
}
```

SELECT DISTINCT should be used by default to eliminate duplicate values in the output variable(s).

5.2.15 Order Output by Variables

Template identifier: order

Parameters:

- ([order1], \dots): directions for variables used for ordering (*order modifier*) ∈ { **ASC**, **DESC** }
- ([var1], \dots): variables used for ordering
- [cont]: query graph to add to

Template 5.21: order, sorting by [var1]

```
[cont]  
ORDER BY [order1] ( [var1] ) [order2] ( [var2] )
```

As indicated in the template, there is an ordering expression for each position of the tuples of the first two parameters. Multiple ordering expressions are added separated by spaces.

This template must have a template with shape **SELECT** \dots **WHERE** { } in its [cont] slot.

5.3 Shortcomings of the Established Templates

5.3.1 Resource-Use of the "connect" Template

If we think of a SPARQL query as a tree, the **connect** template combines two sub-trees (i.e., subqueries) by adding an edge between them. In RDF terms, it takes a variable from one sub-tree as subject and connects it to a variable from another sub-tree, the object, using a predicate. However, using this template may in some cases cause time-outs if very large cartesian products have to be calculated between variables that are yet to be connected by the template. To avoid this, one can use a single query tree where no

new variable is introduced without specifying its relation to the existing variables.

For example, in *Wikipedia Lists* ID 11 (*WL 11*), calculating the cartesian product of cities and countries leads to this problem. It can be solved by first characterizing the cities using **path**, then connecting them to a variable **?country** via the **wdt:P17** property using **add_path**, and then characterizing the country variable using **add_path**. In this variant, one sub-tree is "grown" directly from another.

5.3.2 Arbitrary Containment Relations

Except for the **path** template, all templates have a slot where other instantiated, potentially combined, templates can be inserted. We refer to this as one template containing another. This containment relation may, however, not generally be interpreted as a hierarchical relation indicating the order in which the corresponding query parts are evaluated. In some cases, e.g., when there are multiple instances of **path** and **add_path** in a **WHERE** clause, their order can be swapped arbitrarily without changing the effect of the query.

Most of the devised templates have exactly one slot for other, possibly combined, templates. Because of this, multiple subqueries that are "on the same level" must be realized in a way where one contains the other. This means that variables projected by the subqueries may have to be projected upwards artificially (cf. the parameters with description "additional variables to project"). This causes unnecessary nesting.

5.3.3 Constraints and Expressions in **FILTER** and **BIND** as String Inputs

As can be seen in the template specifications, the constraints and expressions of the **FILTER** and **BIND** templates are input as strings.

In a more fine-grained template-based system, one might implement (sub-)templates that generate the constraints and expressions.

5.4 Further Observations Regarding the Established Templates

5.4.1 Non-Deterministic Usability of Templates

It is important to realize that there are often multiple possibilities to structure a query using the devised templates. For example, instead of applying the **minus** template to several instances of **path** connected by **union**, one could use several instances of **minus**, each negating one **path**.

Similarly, one could use `arg_agg_all` and `arg_ranks_all` as well as `agg_all` and `val_ranks_all` interchangeably if only the top or bottom rank should be considered. When analyzing usage frequencies of templates, this needs to be kept in mind.

5.4.2 Hidden Frequency Differences of Aggregators

Some aggregation operators may be used much more often than others in, e.g., the *agg* template. These frequency differences for different aggregation operators are, however, hidden when counts are computed for the general *agg* template.

For example, **GROUP_CONCAT** is used very frequently to avoid duplicate entities in the output – matching the Wikipedia lists – while operators like **SUM** or **AVG** are much less frequently used. Whether **GROUP_CONCAT** is needed to eliminate duplicates, can only be known once the result set is known. The NL question does not indicate this. In this sense, **GROUP_CONCAT** is a special aggregation operator that may be applied by a KGQA system by default whenever needed. When performing frequency analyses, this is another thing to keep in mind.

5.4.3 Possible Template Additions

limit_offset Template

One could add a `limit_offset` template. Contrary to the *arg_ranks_all* and *val_ranks_all* templates, this template would not consider duplicate values on the specified ranks. Instead, the `limit_offset` template would be used for user inquiries containing requests like "Show me 10 results". In this case, the user would be sure to get at most 10 results – fewer if there are less than 10 results. Using **LIMIT** and **OFFSET**, the template would then allow to slice off the sections of the results that the user does not want to see.

Template Combining agg and arg_agg

It might also be useful to add a template that combines the **agg** and **arg_agg** templates. In *Wikipedia Lists* ID 13, **agg** is applied to sum up the box office values of several movies in each series, to calculate their average box office value, and to count how many movies there are in each series. **arg_agg** is used to determine the highest-grossing movie per series. Since these templates are applied separately, the box office information must be retrieved twice. While few queries seem to require both templates, it might be worthwhile to introduce a combined template for the two types of operations.






6 Usage of Templates

6.1 Example Usage of Templates

This section shows the application of the semantic templates to eight concrete examples. For each example, a tabular overview of its semantic structure called its **semantic plan**, is given. This overview may contain multiple tables which each contain the following information for each semantic characteristic (representing one row in the table):

- its **number**
- a **description** of the semantic characteristic in natural language
- its **semantic category** represented using the symbol and the **semantic template** used to realize it, with filled slots

For convenience, here is a reminder of which symbols represent which semantic category:

ATTRIBUTE	
CONSTRAINT	
AGGREGATE	
COMBINE	
OUTPUT	

For each example, a link to the full query in QLever is provided. The query text was formatted using the [SPARQLer Query Validator](#).

Each example contains at least one new (i.e., not previously shown) template and each template is covered by at least one example.

Example 8 demonstrates that the templates can be used with a knowledge graph other than Wikidata: PubChem.








Some of the examples were taken from benchmarks, while others were self-made. In the former case, their origin is indicated.

6.1.1 Example 1: The city with most museums for each country.

... Show the number of museums and the coordinates of each city and order the results by descending number of museums. [QLever link](#)

The query currently has 841 results – more than there are countries in the world. The large number is in part due to the inclusion of historical countries but also due to the inclusion of ties. For example, some country entities may not have any museums registered in Wikidata. In this case, *all* the cities for this country qualify as the city with the most (i.e., zero) museums in the country.

Table 3: Top-level structure (Example 1)

#	description	semantic category and template
1.	number of museums per city	 Table 4 projecting ?city, ?num_museums
2.	countries per city	 <code>add_path(?city, wdt:P17, ?country, [1])</code>
3.	cities with maximum number of museums per country	 <code>arg_agg(∅, MAX, ?city, ?num_museums, ?max_num_museums, (?country), ∅, [2])</code>
4.	coordinates per city	 <code>add_path(?city, wdt:P625, ?coordinates, [3])</code>
5.	name per city and country	 <code>add_name((?city, ?country), (?city_name, ?country_name), rdfs:label, ("en", "en"), [4])</code>
6.	output city name, country name, maximum number of museums, coordinates	 <code>select(DISTINCT, (?city_name, ?country_name, ?max_num_museums, ?coordinates), [5])</code>
7.	order by descending maximum number of museums	 <code>order((DESC), (?max_num_museums), [6])</code>

Characteristic **2.** contains multiple sub-characteristics and is represented in its own table. When the characteristics are chosen intuitively, the need for sub-characteristics arises naturally. The representation with sub-tables also makes the tables at each level clearer and more concise.

Using the semantic structure, we can, as a first step, transform every characteristic into a piece of syntax. As a second step, we then nest the templates as indicated by the parameters of type **[cont]**. When nesting, if **[cont]** is a subquery, curly braces are added around it.

The `add_name` and `add_desc` templates are said to realize ATTRIBUTES despite containing **FILTER** clauses. This is because we consider the whole template, realizing the semantic characteristic of adding (specific) names or descriptions.

SPARQL 6.1: for Table 3





```

SELECT DISTINCT ?city_name ?country_name ?max_num_museums      # [6]
               ?coordinates
WHERE
{ { SELECT ?city ?country ?max_num_museums                      # [3]
  WHERE
  { { SELECT (MAX(?num_museums) AS ?max_num_museums) ?country
    WHERE
    {
      [1]
      ?city wdt:P17 ?country .                                  # [2]
    } GROUP BY ?country
  }
  { SELECT (?num_museums AS ?max_num_museums) ?city ?country
    WHERE
    {
      [1]
      ?city wdt:P17 ?country .
    }
  }
}
}
?city wdt:P625 ?coordinates .                                  # [4]
?city rdfs:label ?city_name . FILTER(LANG(?city_name)="en")   # [5]
?country rdfs:label ?country_name . FILTER(LANG(?country_name)="en")
}
ORDER BY DESC(?num_museums)                                    # [7]

```

The comments are meant as a help for the reader and do not precisely specify the parts. Instead, they indicate where the parts (e.g., [4] above) *start* to add something new to the query text. In the case of the **UNION** and **MINUS** constructs, the comments are positioned in the line of the operator (cf. Example 2).

Table 4: Partial structure (Example 1): [1] number of museums per city

#	description	semantic category and template
1.1.	cities	 path (?city, wdt:P31/wdt:P279*, wd:Q515)
1.2.	museums	 path (?museum, wdt:P31/wdt:P279*, wd:Q33506)
1.3.	museums located in cities	 connect ([1.1], [1.2], ?museum, wdt:P131 ⁺ , ?city)
1.4.	number of museums per city	 agg ((DISTINCT), (COUNT), (?museum), (?num_museums), (?city), ∅, [1.3], [1.1])

SPARQL 6.2: for Table 4, characteristic [1]

```
SELECT (COUNT(DISTINCT ?museum) AS ?num_museums) ?city      # [1.4]
WHERE
{
  ?city wdt:P31/wdt:P279* wd:Q515 .                          # [1.1]
  OPTIONAL {
    ?city wdt:P31/wdt:P279* wd:Q515 .                        # [1.3], contains \dots
    ?museum wdt:P31/wdt:P279* wd:Q33506 .                   # \dots [1.1] and [1.2]
    ?museum wdt:P131+ ?city .
  }
}
GROUP BY ?city
```

6.1.2 Example 2: How many countries have never been a member of the UN?

Origin: QALD-10, ID 58. The gold query has been modified to use **MINUS** instead of **FILTER NOT EXISTS**.
[QLever link](#)

Since the query is not too complex, it is represented using a single table and a single piece of SPARQL. It shows how the **minus** and **agg_all** templates are used.

Table 5: Structure (Example 2)

#	description	semantic category and template
1.	countries	Q path (?country, wdt:P31/wdt:P279*, wd:Q6256)
2.	UN member entities	Q path (?country, wdt:P463/wdt:P279*, wd:Q1065)
3.	UN non-member countries	Q minus ([1], [2])
4.	number of UN non-member countries	⚙ agg_all ((DISTINCT), (COUNT), (?country), (?num_countries), [3])

SPARQL 6.3: for Table 5

SELECT (COUNT(DISTINCT ?country) AS ?num_countries)	# [4]
WHERE	
{	
{ ?country wdt:P31/wdt:P279* wd:Q6256 }	# [1]
MINUS	# [3]
{ ?country wdt:P463/wdt:P279* wd:Q1065 }	# [2]
}	





6.1.3 Example 3: Which US president was played by the most actors in a movie?

... Show the number of actors and their names. Origin: Wikipedia Lists, ID 2

[Wikipedia List](#) | [QLever link](#)

This example shows how the `arg_agg_all` template is used, and how multiple aggregations are performed using the `agg` template.

Table 6: Top-level structure (Example 3)

#	description	semantic category and template
1.	number of actors that played a US president in a movie per president, and the names of these actors	 Table 7 projecting ?num_actors, ?actor_names, ?president
2.	president(s) with the highest number of actors from 2.	 <code>arg_agg_all</code> (DISTINCT, MAX, ?president, ?num_actors, ?max_num_actors, (?actor_names), [1])
3.	name per president from 3.	 <code>add_name</code> ((?president), (?president_name), rdfs:label, ("en"), [2])
4.	output president name, number of actors, actor names	 <code>select</code> (DISTINCT, (?president_name, ?max_num_actors, ?actor_names), [3])

SPARQL 6.4: for Table 6

```

SELECT DISTINCT ?president_name ?max_num_actors ?actor_names      # [4]
WHERE
{ { SELECT ?president ?president_name ?max_num_actors ?actor_names  # [2]
  WHERE
  {
    [1]
    BIND(MAX(?num_actors) AS ?max_num_actors)
    FILTER(?num_actors=?max_num_actors)
  }
}
?president rdfs:label ?president_name .                          # [3]
FILTER(LANG(?president_name)="en")
}

```

Table 7: Partial structure (Example 3): [1] number of actors that played a US president in a movie per president, and the names of these actors

#	description	semantic category and template
1.1.	US presidents	Q path (?president, wdt:P39, wd:Q11696)
1.2.	actors who played a US president in a movie	Q Table 8 characterizing ?actor, ?actor_name
1.3.	number of actors from 1.2 per president	agg ((DISTINCT, \emptyset), (COUNT, GROUP_CONCAT), (?actor, ?actor_name), (?num_actors, ?actor_names), (?president), \emptyset , [1.2], [1.1])

SPARQL 6.5: for Table 7, characteristic [1]

```

SELECT (COUNT(DISTINCT ?actor) AS ?num_actors) # [1.3]
      (GROUP_CONCAT(DISTINCT ?actor_name; separator="; ") AS ?actor_names)
WHERE
{
  ?president wdt:P39 wd:Q11696 . # [1.1]
  OPTIONAL
  {
    [1.2]
  }
}
GROUP BY ?president

```

Note that US presidents are characterized in both [1.1] and [1.2.1] because in this query, the triple appears within the **OPTIONAL** and non-**OPTIONAL** sections of the **agg** template using **COUNT**. However, it is only *obligatory* in the non-**OPTIONAL** section.

SPARQL 6.6: for Table 8, characteristic [1.2]

```

?president wdt:P39 wd:Q11696 . # [1.2.1]
?president ^pq:P453 ?p161 . # [1.2.2]
?actor wdt:P106/wdt:P279* wd:Q33999 . # [1.2.3]
?p161 ps:P161 ?actor . # [1.2.4]
?movie wdt:P31/wdt:P279* wd:Q11424 . # [1.2.5]
?p161 ^p:P161 ?movie . # [1.2.6]
?actor rdfs:label ?actor_name . # [1.2.7]
FILTER(LANG(?actor_name)="en")

```

Table 8: Partial structure (Example 3): [\[1.2\]](#) actors who played a US president in a movie

#	description	semantic category and template
1.2.1.	US presidents	Q path (?president, wdt:P39, wd:Q11696)
1.2.2.	statement nodes with qualifier "has role of" with value president (nodes)	Q add__path (?president, ^pq:P453, ?p161, [1.2.1])
1.2.3.	actors	Q path (?actor, wdt:P106/wdt:P279*, wd:Q33999)
1.2.4.	value per node is actor	Q connect ([1.2.2], [1.2.3], ?p161, ps:P161, ?actor)
1.2.5.	movies	Q path (?movie, wdt:P31/wdt:P279*, wd:Q11424)
1.2.6.	subject of node is movie	Q connect ([1.2.4], [1.2.5], ?p161, ^p:P161, ?movie)
1.2.7.	name per actor	Q add__name ((?actor), (?actor__name), rdfs:label, ("en"), [1.2.6])

6.1.4 Example 4: Famous twins

... Show each twin with their twin sibling (if available), as well as with both people's names, descriptions and date of birth. Origin: Wikipedia Lists, ID 9 | [Wikipedia list](#) | [QLever link](#)








The query shows how some queries can become very complex when they combine *directly* accessible information (here: using the "twin" class) with *indirectly* accessible information (here: twins beings characterized as people with a biological sibling born on the same date).

In general, it is possible for a twin to occur both as `?twin` and `?twin_2` in the output. Moreover, in the case of triplets, quadruplets, etc., the same person will even occur multiple times as `?twin`, and multiple times as `?twin_2`.

Besides, due to their twofold characterizations, some `?twin` entities appear twice in the results list. One can deal with this by using **GROUP_CONCAT** (cf. the benchmark queries for this example, WL 9).

The example shows the use of the `optional`, `add_desc`, `filter` and `union` templates.

Table 9: Top-level structure (Example 4)

#	description	semantic category and template
1.	twins with birthdate and name and description of twin sibling if available	 Table 10 characterizing ?twin, ?date_of_birth, ?twin_2_name, ?twin_2_desc
2.	twins are humans	 <code>add_path(?twin, wdt:P31/wdt:P279*, wd:Q5, [1])</code>
3.	description per twin	 <code>add_desc((?twin), (?twin_desc), schema:description, ("en"), [2])</code>
4.	name per twin	 <code>add_name((?twin), (?twin_name), rdfs:label, ("en"), [3])</code>
5.	number of site links per twin	 <code>add_path(?twin, ^schema:about/wikibase:sitelinks, ?sitelinks, [4])</code>
6.	output twin name, twin description, date of birth, twin sibling name, twin sibling description	 <code>select(DISTINCT, (?twin_name, ?twin_desc, ?date_of_birth, ?twin_2_name, ?twin_2_desc), [5])</code>
7.	order by descending number of site links of twin to show famous twins first	 <code>order((DESC), (?sitelinks), [6])</code>

As mentioned before, the query retrieves people who are characterized as twins via the "twin" item as well as via familial relations. For the people characterized via the "twin" item it is not guaranteed that information about their twin sibling will be available. This is covered by the **union** template used in [1].

SPARQL 6.7: for Table 9

```

SELECT DISTINCT ?twin_name ?twin_desc ?date_of_birth           # [6]
                  ?twin_2_name ?twin_2_desc
WHERE
{
  [1]
  ?twin wdt:P31/wdt:P279*                                     # [2]
  ?twin schema:description ?twin_desc .                      # [3]
  FILTER(LANG(?twin_desc)="en")
  ?twin rdfs:label ?twin_name .                                # [4]
  FILTER(LANG(?twin_name)="en")
  ?twin ^schema:about/wikibase:sitelinks ?sitelinks .        # [5]
}
ORDER BY DESC(?sitelinks)                                     # [7]

```

Table 10: Partial structure (Example 4): [1] twins with birthdate, and name and description of twin sibling if available

#	description	semantic category and template
1.1.	twin (using type "twin")	🔍 path (?twin, wdt:P31/wdt:P279*, wd:Q159979)
1.2.	birthdate per twin from 1.1	🔍 add__path (?twin, wdt:P569, ?date_of_birth, [1.1])
1.3.	twins (characterized indirectly) with name and description of twin sibling and shared birthdate	🔍 Table 11 characterizing ?twin, ?date_of_birth, ?twin_2_name, twin_2_desc
1.4.	twins with birthdate, and name and description of twin sibling if available	★ union (([1.2], [1.3]), ∅)

SPARQL 6.8: for Table 10, characteristic [1]

```
{
  ?twin wdt:P31/wdt:P279* wd:Q159979 .           # [1.1]
  ?twin wdt:P569 ?date_of_birth .                 # [1.2]
}
UNION                                             # [1.4]
{ [1.3] }
```

The following structure is used to characterize people as twins, using the fact that they have a biological sibling born on the same date.

1.2.1 contains sub-characteristics because we need to ensure the precision of at least 11 for the date of birth, to use "complete" birthdates, containing the day, month, and year.

1.2.2 contains sub-characteristics because we combine mothers characterized via the "mother" property as well as mothers characterized via the "parent" property, as well as via the "sex or gender" property and the "female" item.

Table 11: Partial structure (Example 4): [1.3], twins (characterized indirectly) with name and description of twin sibling and shared birthdate

#	description	semantic category and template
1.3.1.	date of birth with precision 11 or higher per twin	⊕ Table 12 characterizing ?date_of_birth
1.3.2.	mother per twin	⊕ Table 13 characterizing ?mother
1.3.3.	children per mother (twin siblings)	⊕ add_path (?mother, wdt:P40, ?child, [1.3.2])
1.3.4.	twin sibling has same birthday as twin	⊖ add_path (?child, wdt:P569, ?date_of_birth, [1.3.3])
1.3.5.	twin sibling is different from twin	⊖ filter ("?twin != ?child", [1.3.4])
1.3.6.	name per twin sibling	⊕ add_name ((?twin_2), (?twin_2_name), rdfs:label, ("en"), [1.3.5])
1.3.7.	description per twin sibling	⊕ add_desc ((?twin_2), (?twin_2_desc), schema:description, ("en"), [1.3.6])

SPARQL 6.9: for Table 11, characteristic [1.3]

```
{
  [1.3.1]
  [1.3.2]
  ?mother wdt:P40 ?twin_2 .                # [1.3.3]
  ?twin_2 wdt:P569 ?date_of_birth .         # [1.3.4]
}
FILTER(?twin != ?twin_2)                   # [1.3.5]
?twin_2 rdfs:label ?twin_2_name .          # [1.3.6]
FILTER(LANG(?twin_2_name)="en")
?twin_2 schema:description ?twin_2_desc .  # [1.3.7]
FILTER(LANG(?twin_2_desc)="en")
```

The indirect characterization of twins starts by retrieving a new set of items with a sufficiently precise value for "date of birth":

Table 12: Partial structure (Example 4): [1.3.1] date of birth per twin with precision 11 or higher

#	description	semantic category and template
1.3.1.1.	date of birth node per twin (nodes)	🔍 path (?twin, p:P569, ?p569)
1.3.1.2.	node has best rank	🔍 add__path (?p569, rdf:type, wikibase:BestRank, [1.3.1.1])
1.3.1.3.	date of birth value node per node (value nodes)	🔍 add__path (?p569, psv:P569, ?psv569, [1.3.1.2])
1.3.1.4.	time precision per value node	🔍 add__path (?psv569, wikibase:timePrecision, ?time__precision, [1.3.1.3])
1.3.1.5.	time precision at least 11	🔍 filter ("?time__precision >= 11", [1.3.1.4])
1.3.1.6.	time value per value node	🔍 add__path (?psv569, wikibase:timeValue, ?date_of__birth, [1.3.1.5])

SPARQL 6.10: for Table 12, characteristic [1.3.1]

```
{
  ?twin p:P569 ?p569 . # [1.3.1.1]
  ?p569 rdf:type wikibase:BestRank . # [1.3.1.2]
  ?p569 psv:P569 ?psv569 . # [1.3.1.3]
  ?psv569 wikibase:timePrecision ?time_precision . # [1.3.1.4]
}
FILTER(?time_precision >= 11) # [1.3.1.5]
?psv569 wikibase:timeValue ?date_of_birth . # [1.3.1.6]
```

Table 13: Partial structure (Example 4): [1.3.2] mother per twin

#	description	semantic category and template
1.3.2.1.	mother per twin (using property "mother")	Q path(?twin, wdt:P25, ?mother)
1.3.2.2.	parent per twin	Q path(?twin, ^wdt:P40, ?mother)
1.3.2.3.	parent is female	Q add_path(?mother, wdt:P21, wd:Q6581072, [1.3.2.2])
1.3.2.4.	mother per twin	★ union(([1.3.2.1], [1.3.2.3]), [1.3.1])

In addition to using the "child" property, one could use the "son" and "daughter" properties, but this was not done to avoid further complexity.

SPARQL 6.11: for Table 13, characteristic [1.3.2]

```
{ ?twin wdt:P25 ?mother } # [1.3.2.1]
UNION # [1.3.2.4]
{
  ?twin ^wdt:P40 ?mother . # [1.3.2.2]
  ?mother wdt:P21 wd:Q6581072 . # [1.3.2.3]
}
```

6.1.5 Example 5: List of countries whose capital is not their largest city

...Show the country flag image, the country name, capital name, largest city name, the population for both cities, and the ratio between the populations.

Origin: Wikipedia Lists, ID 11 | [Wikipedia list](#) | [QLever link](#)

The version of the query presented here is a simplified version, using some shorter variable names and leaving out some CONSTRAINTs to reduce complexity. The characteristics that were left out are listed at the end of this subsection.

This example shows how the **bind** template is used and includes an instance of the **arg_agg** template with grouping by multiple variables. It also contains an instance of the **agg_all** template with grouping by multiple variables.

Table 14: Top-level structure (Example 5)

#	description	semantic category and template
1.	countries whose capital is not their largest city	🔍 Table 15 characterizing ?capital, ?city (largest city), ?city_pop
2.	flag image per country from 1.	🔍 add_path (?country, wdt:P41, ?flag_image, [1])
3.	population per capital from 1.	🔍 add_path (?capital, wdt:P1082, ?capital_pop, [2])
4.	ratio of largest city's population and capital population	★ bind ("?city_pop / ?capital_pop", ?pop_ratio, [3])
5.	name per country, capital and largest city	🔍 add_name ((?country, ?capital, ?city), (?country_name, ?capital_name, ?city_name), rdfs:label, ("en", "en", "en"), [4])
6.	output flag image, country name, capital name, capital population, (largest) city name, (largest) city population, population ratio	👁 select (DISTINCT, (?country_name, ?capital_name, ?capital_pop, ?city_name, ?max_city_pop, ?pop_ratio), [5])

SPARQL 6.12: for Table 14








```

SELECT DISTINCT ?flag_image ?country_name ?capital_name      # [6]
                ?capital_pop ?city_name ?max_city_pop ?pop_ratio
WHERE
{
  [1]
  ?country wdt:P41 ?flag_image .                             # [2]
  ?capital wdt:P1082 ?capital_pop .                          # [3]
  BIND((?max_city_pop/?capital_pop) AS ?pop_ratio)           # [4]
  ?country rdfs:label ?country_name .                        # [5]
  FILTER(LANG(?country_name)="en")
  ?capital rdfs:label ?capital_name .
  FILTER(LANG(?capital_name)="en")
  ?city rdfs:label ?city_name .
  FILTER(LANG(?city_name)="en")
}

```

Due to heavy nesting, this partial structure and query is best displayed as a whole, including further sub-characteristics:

Table 15: Partial structure (Example 5): [1] countries whose capital is not their largest city

#	description	semantic category and template
1.1	countries	 path (?country, wdt:P31/wdt:P279*, wd:Q6256)
1.2.	cities	 path (?city, wdt:P31/wdt:P279*, wd:Q515)
1.3.	cities are in countries	 join ([1.1], [1.2], ?city, wdt:P17, ?country)
1.4.	population per city	 add_path (?city, wdt:P1082, ?city_pop, [1.3])
1.5.	cities with maximum population per country	 arg_agg (\emptyset , MAX, ?city, ?city_pop, ?max_city_pop, (?country), \emptyset , [1.4])
1.6.	capital per country	 add_path (?country, wdt:P36, ?capital, [1.5])
1.7.	capitals unequal to largest cities	 filter ("?city != ?capital", [1.6])

SPARQL 6.13: for Table 15, characteristic [1]

```
{ { SELECT ?city ?country ?max_city_pop           # [1.5]
  WHERE
    { { SELECT (MAX(?city_pop) AS ?max_city_pop) ?country
      WHERE
        { ?country wdt:P31/wdt:P279* wd:Q6256 .      # [1.1]
          ?city wdt:P31/wdt:P279* wd:Q515 .          # [1.2]
          ?city wdt:P17 ?country .                  # [1.3]
          ?city wdt:P1082 ?city_pop .                # [1.4]
        }
      GROUP BY ?country
    }
    { SELECT (?city_pop AS ?max_city_pop) ?city ?country
      WHERE
        { ?country wdt:P31/wdt:P279* wd:Q6256 .
          ?city wdt:P31/wdt:P279* wd:Q515 .
          ?city wdt:P17 ?country .
          ?city wdt:P1082 ?city_pop
        }
      }
    }
  }
}
?country wdt:P36 ?capital .
}                                     # [1.6]
FILTER(?capital != ?city)           # [1.7]
```

Note that one could do all of the following things to avoid the inclusion of historical and unrecognized countries, and to eliminate duplicate values that arise when using **wdt**:

- exclude instances of
 - "historical country"
 - "state with limited recognition"
 - "micronation"
- use the most recent capital, i.e., the one with the maximum "start time" qualifier value, and no "end time" qualifier value
- use the most recent population value, i.e., the one with the maximum "point in time" qualifier value

Since the question does not specifically require these things, these steps are left out here.

6.1.6 Example 6: What is the combined total revenue of the three largest Big Tech companies ordered by number of employees?

... Use the most recent values for the total revenue and the number of employees. Origin: QALD-10, ID 203 | [QLever link](#)

Modifications to QALD-10, ID 203:

The words "most recent" were added to the NL question twice for clarity.

Due to using the templates, the query constructed here accounts for ties regarding the "three largest Big Tech companies". Moreover, the query uses the most recent available value for both the number of employees and the total revenue, avoiding potential multiple values that could arise when using **wdt**. While the QALD gold query also uses the most recent total revenue value, it retrieves the numbers of employees with **wdt**.



This query is very long (132 lines including prefixes) due to the **arg_agg** and **arg_ranks_all** templates that both contain duplicate code.

The distinctive naming of the variable pairs **?time** and **?max_time**, and **?time_2** and **?max_time_2** is not necessary due to them having different scopes. However, it is done to avoid confusion and to indicate that the variables don't share a scope.

While there is a Q-item "Big Tech", companies like Microsoft are only registered as "part of" the "Big Tech (web)". As such, the latter item was used.

This example shows the application of the **arg_ranks_all** template.






Table 16: Top-level structure (Example 6)

#	description	semantic category and template
1.	most recent total revenue per Big Tech company with number of employees on ranks 1 through 3 (descending order)	 Table 17 projecting ?company, ?total_revenue, ?max_time
2.	sum of total revenues from 1.	 agg_all ((\emptyset), (SUM), (?total_revenue), (?sum_total_revenue), [1])

SPARQL 6.14: for Table 16

```
SELECT (SUM(?total_revenue) AS ?sum_total_revenue) # [2]
WHERE
{
  [1]
}
```

Table 17: Partial structure (Example 6): [1] most recent total revenue per Big Tech company with number of employees on ranks 1 through 3 (descending order)

#	description	semantic category and template
1.1.	Big Tech companies with number of employees on ranks 1 through 3 (descending order)	 Table 18 projecting ?company
1.2.	"total revenue" statement node per company (nodes)	 <code>add_path(?company, p:P2139, ?p2139, [1.1])</code>
1.3.	value per node (total revenue)	 <code>add_path(?p2139, ps:P2139, ?total_revenue, [1.2])</code>
1.4.	point in time per total revenue	 <code>add_path(?p2139, pq:P585, ?time, [1.3])</code>
1.5.	most recent total revenue per company	 <code>arg_agg(∅, MAX, ?total_revenue, ?time, ?max_time, (?company), ∅, [1.4])</code>







SPARQL 6.15: for Table 17, characteristic [1]

```

SELECT ?total_revenue ?company ?max_time # [1.5]
WHERE
{ { SELECT (MAX(?time) AS ?max_time) ?company
  WHERE
  {
    [1.1]
    ?company p:P2139 ?p2139 . # [1.2]
    ?p2139 ps:P2139 ?total_revenue . # [1.3]
    ?p2139 pq:P585 ?time . # [1.4]
  }
  GROUP BY ?company
}
{ SELECT (?time AS ?max_time) ?total_revenue ?company
  WHERE
  {
    [1.1]
    ?company p:P2139 ?p2139 .
    ?p2139 ps:P2139 ?total_revenue .
    ?p2139 pq:P585 ?time .
  }
}
}

```

Table 18: Partial structure (Example 6): [1.1] Big Tech companies with number of employees on ranks 1 through 3 (descending order)

#	description	semantic category and template
1.1.1.	Big Tech companies	 path(?company, wdt:P361, wd:Q30748112)
1.1.2.	"employees" statement node per company (nodes)	 add_path(?company, p:P1128, ?p1128, [1.1.1])
1.1.3.	value per node (number of employees)	 add_path(?p1128, ps:P1128, ?employees, [1.1.2])
1.1.4.	point in time per node	 add_path(?p1128, pq:P585, ?time_2, [1.1.3])
1.1.5.	most recent number of employees per company	 arg_agg(∅, MAX, ?employees, ?time_2, ?max_time_2, (?company), ∅, [1.1.4])
1.1.6.	Big Tech companies with number of employees on ranks 1 through 3 (descending order)	 arg_ranks_all(?company, DESC, ?employees, 3, 0, ∅, [1.1.5])

SPARQL 6.16: for Table 18, characteristic [1.1]

```
SELECT ?company # [1.1.6]
WHERE
{ { SELECT ?company ?employees
  WHERE
  { SELECT ?employees ?company ?max_time_2 # [1.1.5]
    WHERE
    { { SELECT (MAX(?time_2) AS ?max_time_2) ?company
      WHERE
      { ?company wdt:P361 wd:Q30748112 . # [1.1.1]
        ?company p:P1128 ?p1128 . # [1.1.2]
        ?p1128 ps:P1128 ?employees . # [1.1.3]
        ?p1128 pq:P585 ?time_2 . # [1.1.4]
      }
    }
    GROUP BY ?company
  }
  { SELECT (?time_2 AS ?max_time_2) ?employees ?company
    WHERE
    { ?company wdt:P361 wd:Q30748112 .
      ?company p:P1128 ?p1128 .
      ?p1128 ps:P1128 ?employees .
      ?p1128 pq:P585 ?time_2 .
    }
  }
}
}
}
ORDER BY DESC(?employees)
OFFSET 0
LIMIT 3
}
{ SELECT ?company ?employees
  WHERE
  { SELECT ?employees ?company ?max_time
    WHERE
    { { SELECT (MAX(?time_2) AS ?max_time_2) ?company
      WHERE
      { ?company wdt:P361 wd:Q30748112 .
        ?company p:P1128 ?p1128 .
        ?p1128 ps:P1128 ?employees .
        ?p1128 pq:P585 ?time_2 .
      }
    }
    GROUP BY ?company
  }
  { SELECT (?time_2 AS ?max_time_2) ?employees ?company
    WHERE
    { ?company wdt:P361 wd:Q30748112 .
      ?company p:P1128 ?p1128 .
      ?p1128 ps:P1128 ?employees .
      ?p1128 pq:P585 ?time_2 .
    }
  }
}
}
}
}
```

6.1.7 Example 7: How many years did the second oldest dog in the world live?

Origin: QALD-10, ID 119 | [QLever link](#)

While the gold query retrieves entities that are instances of the "dog" class using **wdt:P31**, the query constructed here includes entities of type "dog" that are instances of sub-classes of the "dog" class using **wdt:P31/wdt:P279***. It includes ~20 additional dogs. The retrieval of the age of the dogs seems to successfully exclude fictional dogs.

This example shows the use of the **val_ranks_all** template.

Table 19: Structure (Example 7)

#	description	semantic category and template
1.	dogs	Q path (?dog, wdt:P31/wdt:P279*, wd:Q144)
2.	date of birth per dog	+ add_path (?dog, wdt:P569, ?date_of_birth, [1])
3.	date of death per dog	+ add_path (?dog, wdt:P570, ?date_of_death, [2])
4.	approximate age per dog	★ bind ("((YEAR(?date_of_death) - YEAR(?date_of_birth)) - IF(((MONTH(?date_of_death) < MONTH(?date_of_birth)) ((MONTH(?date_of_death) = MONTH(?date_of_birth)) && (DAY(?date_of_death) < DAY(?date_of_birth))))), 1, 0))", ?age, [3])
5.	age value on rank 2	Q val_ranks_all (DESC, ?age, 1, 1, ∅, [4])

SPARQL 6.17: for Table 19

```
SELECT ?age
WHERE
{
    ?dog wdt:P31/wdt:P279* wd:Q144 .           # [5]
    ?dog wdt:P569 ?date_of_birth .             # [1]
    ?dog wdt:P570 ?date_of_death .             # [2]
    BIND(((YEAR(?date_of_death) - YEAR(?date_of_birth))
        - IF(((MONTH(?date_of_death) < MONTH(?date_of_birth))
            || ((MONTH(?date_of_death) = MONTH(?date_of_birth))
            && (DAY(?date_of_death) < DAY(?date_of_birth))))), 1, 0)) AS ?age) # [3]
                                                # [4]
}
ORDER BY DESC(?age) LIMIT 1 OFFSET 1
```

6.1.8 Example 8: NSAID compounds with molecular weight < 200 g/mol

Origin: QLever example query for the PubChem backend | [QLever link](#)

This example illustrates how the templates may be applied when using a knowledge base other than Wikidata. In the above example, the class of NSAIDs is not available as a pre-made item but has to be defined as a **restriction class**, as described in the beginning of Chapter 5.

The example highlights how PubChem stores some information in a **key-value structure**: The attribute of "molecular weight" is characterized by the key "molecular weight calculated by the PubChem software library" ([2.1.2]); its value is accessed afterward ([2.2.1]).

Table 20: Top-level structure (Example 8)

#	description	semantic category and template
1.	NSAID compounds with molecular weight smaller than 200 g/mol	🔍 Table 21 characterizing ?nsaid, ?compound, ?mol_weight_val
2.	name per NSAID	🔍 add_name ((?nsaid), (?nsaid_name), rdfs:label, (), [1])
3.	output compound, NSAID name	👁 select (DISTINCT, (?compound, ?nsaid_name), [2])

SPARQL 6.18: for Table 20

```
SELECT DISTINCT ?compound ?nsaid_name # [3]
WHERE
{
  [1]
  ?nsaid rdfs:label ?nsaid_label . # [2]
}
```

Table 21: Partial structure (Example 8): [1] NSAID compounds with molecular weight smaller than 200 g/mol. Show names of NSAIDs.

#	description	semantic category and template
1.1.	NSAIDs (non-steroidal anti-inflammatory drugs)	Q Table 22 characterizing ?nsaid
1.2.	NSAID compounds	⊕ add_path(?nsaid, ^rdf:type, ?compound, [1.1])
1.3.	attributes per NSAID compound	⊕ add_path(?compound, sio:SIO_000008, ?mol_weight_attr, [1.2])
1.4.	molecular weight attribute per NSAID compound	Q add_path(?mol_weight_attr, rdf:type, sio:CHEMINF_000334, [1.3])
1.5.	molecular weight value per NSAID compound	⊕ add_path(?mol_weight_attr, sio:SIO_000300, ?mol_weight_val, [1.4.])
1.6.	molecular weight value is smaller than 200	Q filter("?mol_weight_val < 200", [1.5])

SPARQL 6.19: for Table 21, characteristic [1]

```

{
  [1.1]
  ?compound ^rdf:type ?nsaid .           # [1.2]
  ?compound sio:SIO_000008 ?mol_weight_attr . # [1.3]
  ?mol_weight_attr rdf:type sio:CHEMINF_000334 . # [1.4]
  ?mol_weight_attr sio:SIO_000300 ?mol_weight_val . # [1.5]
}
FILTER(?mol_weight_val < 200)           # [1.6]

```

SPARQL 6.20: for Table 22, characteristic [1.1]

```

?nsaid rdfs:subClassOf ?class .           # [1.1.1]
?class rdf:type owl:Restriction .        # [1.1.2]
?class owl:onProperty obo:RO_0000087 .   # [1.1.3]
?class owl:someValuesFrom obo:CHEBI_35475 . # [1.1.4]

```


Table 22: Partial structure (Example 8): **[1.1]** NSAIDs (non-steroidal anti-inflammatory drugs)

#	description	semantic category and template
1.1.1.	NSAIDs sub-class of to-be-defined class	Q path (?nsaid, rdfs:subClassOf, ?class)
1.1.2.	class is restriction class	Q add_path (?class, rdf:type, owl:Restriction, [1.1.1])
1.1.3.	class refers to role	Q add_path (?class, owl:onProperty, obo:RO_0000087, [1.1.2])
1.1.4.	class has value non-steroidal anti-inflammatory drug	Q add_path (?class, owl:someValuesFrom, obo:CHEBI_35475, [1.1.3])

6.2 Analysis of Semantic Alternatives

In this section, we briefly explore which semantic purposes the templates perform, using the semantic categories introduced in Chapter 5 and the eight examples just presented.

This analysis is performed **at the level of the templates**. If we, for example, consider people (variable `?person`) and want to add the information who their parents are (`?parent`) using the "mother" and "father" properties, the **union** template is used for that. By itself, this template does not realize an ATTRIBUTE. Instead, it realizes the category "COMBINE" by uniting two characterizations of the `?parent` variable. The triples in the **union** template are instances of **path** that each realize a CONSTRAINT by defining the new variables `?person` and `?parent`.

The counts are shown once for only the Wikidata-based examples (Examples 1–7, "**WD**") and once for all examples, including the PubChem example (Example 8, "**WD & PC**").

The counts are not indicative of the true frequency distributions, but they can provide some insights into which templates are able to fulfill which semantic roles.

6.2.1 Q ATTRIBUTE

	add_path	add_name	add_desc
WD	21	4	1
WD & PC	24	5	1

All examples except for Example 2 (Section 6.1.2) contain at least one ATTRIBUTE.

6.2.2 Q CONSTRAINT

	path	add_path	connect	filter	minus	arg_ranks_all	val_ranks_all
WD	16	4	4	3	1	1	1
WD & PC	17	8	4	4	1	1	1

All examples include a template realizing a CONSTRAINT as this is needed for a non-empty output.

6.2.3 🏗️ AGGREGATE

	agg	agg_all	arg_agg	arg_agg_all
WD	2	2	4	1
WD & PC	2	2	4	1

Most of the examples contain aggregation. However, it must be taken into account that the examples were chosen such that all templates, including aggregation templates, were covered.

6.2.4 ★ COMBINE

	union	bind
WD	2	2
WD & PC	2	2

Only a few examples contain templates realizing COMBINES. In real queries, rather than example queries, **union** is sometimes used extensively to deal with structural inconsistencies, making it appear more often.

6.2.5 👁️ OUTPUT

	select	order
WD	5	2
WD & PC	5	2

Three of the eight queries contain neither a **select** nor a **order** template as they each return a single numeric value.

6.2.6 Conclusions

Almost all templates appear to realize exactly one semantic category. The only exception is **add_path**, which – according to this small dataset – realizes an **ATTRIBUTE** 75% of the time and a **CONSTRAINT** 25% of the time. Cf. the discussion about its semantic purpose in Section 5.2.1.

The **CONSTRAINT** category appears to be the most complex one, containing seven different templates to realize it.

6.3 Analysis of Template Usage

6.3.1 Template Counts

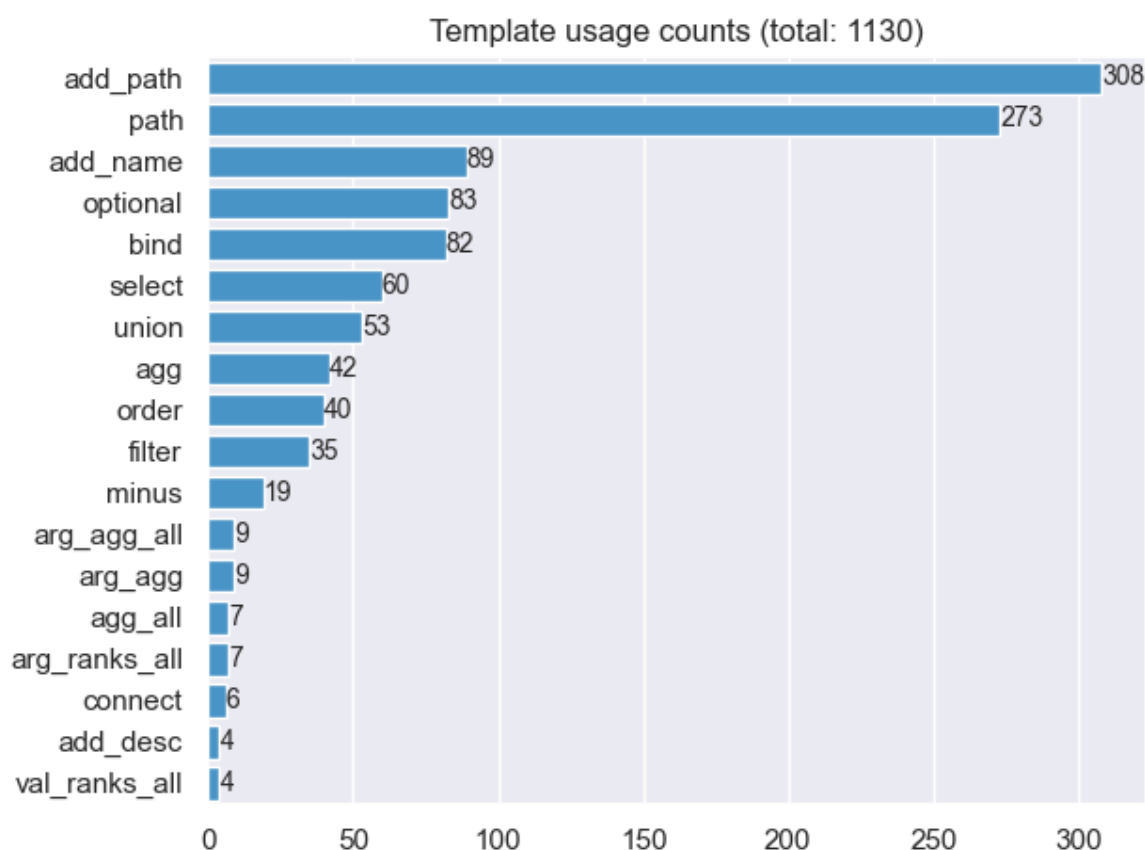
Among the 1,130 template instances, ~51% are triple patterns of either type `path` or `add_path` (cf. Figure 3). The next most frequent template, `add_name`, only makes up ~8% of the instances.

However, it must be noted that the high number of the triple patterns is in part due to code repetition in certain templates: `arg_agg`, `arg_ranks_all`, `agg` when used with `COUNT` while using the whole non-`OPTIONAL` part as the `OPTIONAL` part.

Taking this into account, the differences should be a little less dramatic.

Since `select` was used with all examples to receive distinct results, and exactly once, its count is 60. Two-thirds of the examples used a custom order with `order`.

Figure 3: Template counts for *Wikipedia Lists*



Among the aggregation templates, **agg** occurs the most often, making up ~4% of all template instances. A lot of this is due to **GROUP_CONCAT** being used while eliminating duplicate values. One-third of the instances of **agg** contain an aggregation with **GROUP_CONCAT** as their only aggregation.

Non-surprisingly, **add_name** is used often to produce human-readable output.

optional ensures that entities are output despite having missing attribute values. **union** is used to capture structural inconsistencies.

bind is used often in the benchmark, mostly because of conversions between values in meters and feet which require the template once for performing calculations, and once for choosing the best value with **COALESCE** (i.e., preferring directly available values over calculated ones). These instances account for ~29% of all **bind** instances.

Another frequent use of **bind** is the extraction of the year part of a date, accounting for ~25% of all **bind** instances. Calculations other than those used for unit conversions, with and without parts of date literals, account for (~22%). Less frequent uses of the template are the formatting of result commons with **CONCAT** (~12%), and uses of **COALESCE** (~10%) to preferably use one variable's value over another's outside of unit conversions.

filter is mostly used to apply constraints on the ranges of numerical or date values using comparison operators like $>$ or \leq (~66% of all **filter** instances).

The remaining instances are in roughly equal parts instances using the following:

- **!=** to ensure unequal variable values, e.g., **?var1 != ?var2**
- **=** combined with **YEAR**, e.g., **YEAR(?marriage)=1990**
- **REGEX**, e.g., **REGEX(?description, "canceled")**

6.3.2 Template Containment Relations

This section shows how often the instances of the different template types contained one another in the *Wikipedia Lists* benchmark. Due to the large number of templates, templates that are exclusively found inside **WHERE** clauses were grouped together as the group "where". The full heatmap, without grouping, is found in the appendix.

It comes as no surprise that the **order** template exclusively "contains" the **select** template, as **select** was applied to all benchmark examples before an optional ordering.

select contains **agg** fairly often (~32% of the time), in large part due to the use of **GROUP_CONCAT**.

Figure 4: Template containment relations with grouping for *Wikipedia Lists*

	Contained template							
	where	select	agg	arg_agg_all	arg_agg	arg_ranks_all	agg_all	val_ranks_all
where	860	0	13	9	7	6	2	1
agg	52	0	0	0	1	0	0	0
order	0	40	0	0	0	0	0	0
select	32	0	20	0	0	1	5	2
arg_agg	18	0	0	0	0	0	0	0
arg_ranks_all	8	0	4	0	2	0	0	0
agg_all	6	0	0	0	0	0	0	1
arg_agg_all	4	0	5	0	0	0	0	0
val_ranks_all	2	0	2	0	0	0	0	0

Containment Relations of Subquery-Inducing Templates

Apart from `arg_agg`, all subquery-inducing templates (i.e., aggregation and ranks templates) contained other subquery-inducing templates. These combinations produce highly nested queries. Since the resulting, nested queries are structurally challenging and also "high-level" enough to be indicated by aspects of the NL question, we analyze them here.

For each combination, we list the NL question if available, highlighting parts that might point to the combinations being needed in boldface.

Note that instances of templates being contained by `arg_ranks_all` each contribute two counts instead of one to the heatmap. This is due to the code duplication in `arg_ranks_all`.

- `agg` containing `arg_agg`:

This containment relation is artificial and non-informative with both variables logically being on the same level and using different variables.

- `agg_all` containing `val_ranks_all`:

- "How many inhabitants did the world's three largest cities (by population in the year 2018) have combined?"

- `arg_agg_all` containing `agg`

- "Which US president was played by the most actors in a movie? Also show the actors"
- "Which Formula One driver won the most championships and in which years?"
- "Which movie has won the most Oscars?"
- "Who composed the music for the most Pixar films (excluding short films)?"
- "Which country borders the most other countries?"

- `val_ranks_all` containing `agg`:

This containment relation is due to grouping to eliminate duplicates for the entities whose attribute values are ranked (cf. Section 5.2.5).

- `arg_ranks_all` containing `agg`:

- No NL question available. The templates are used to retrieve the top 10 languages – more than 10 in the case of ties – ranked by the number of countries using them as an official language.
- "Which are the top 10 countries and territories with the highest average elevation?"

- `arg_ranks_all` containing `arg_agg`

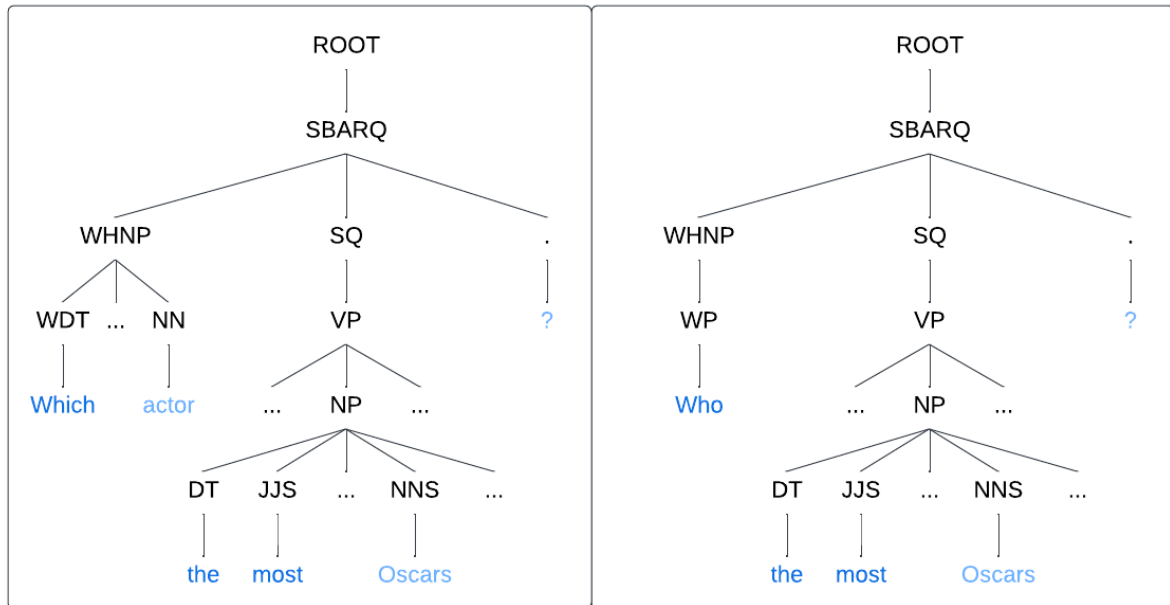
- No NL question available. The templates are used to retrieve the top 50 TikTokers – more than 50 in the case of ties – ranked by their most recent follower count.

In the case of `arg_agg_all` containing `agg`, there is enough data to show a clear pattern.

To define this pattern, the NL questions of the corresponding examples were parsed using the Stanford Lexicalized Parser v2.0.4. [33], using the English PCFG (probabilistic context-free grammar) model. This parser outputs tags from the Penn Treebank syntactic tagset and POS tagset.

Figure 5 shows the syntactic structure that was identified. Fixed, obligatory parts of it are represented in dark blue while light blue parts represent exemplary realizations of required parts. Parts marked "..." represent optional syntactic extensions. As can be seen, there are two versions in the WHNP part. With more data, one could further refine the structure.

Figure 5: Syntactic structure of NL questions that may correspond to queries with `arg_agg_all` containing `agg`



In all the examples, `arg_agg_all` uses `MAX`, and the contained `agg` uses `COUNT` as its aggregator.

Importantly, encountering this structure in an NL question does not mean that we need to use this template combination. Consider the example "Which country has the most inhabitants?" where we would employ `arg_agg_all`, but would not use `agg`. Instead, the number of inhabitants per country can be retrieved using `add_path` and the "population" property: `?country wdt:P1082 ?inhabitants .`

Whenever the number of something is a ready-made property in the KG, its maximum value can be retrieved without using **COUNT**. Other examples would be "Which building has the most (above-ground) floors?" or "Who has the most children?", using the properties "floors above ground" and "number of children" respectively.

There might also be examples using **SUM** instead of **COUNT** in **agg**, e.g., "Who is the most prolific author in history by word count?", where we would have to sum up the "number of words" values of the author's works.

Regarding the other combinations of subquery-inducing templates, there is too little data to infer any patterns. We can however formulate certain hypotheses:

- the word "combined" indicates that an aggregation template with **SUM** is needed
- the word "average" indicates that an aggregation template with **AVG** is needed
- the structure "CD JJS" – a cardinal number followed by an adjective in the superlative, such as "three largest" – indicates that either **val_ranks_all** or **arg_ranks_all** is needed; the latter being indicated by "WDT" (e.g., "which", "what" being used as determiners).
- the word "top" followed by a cardinal number indicating one of the ranking templates

6.3.3 Conclusions

One finding from this chapter is that triple patterns make up about half of the templates when re-creating Wikipedia lists and asking questions about them.

One reason for this may be the structural inconsistency in Wikidata, which also accounts for **union** being used quite a lot. If a ground truth like a Wikipedia list is used, it is easy to uncover structural inconsistencies: Entities missing from the QLever output are analyzed (i.e., their Wikidata pages) and other properties that are used to store the same information are discovered. For example, in *WL 15*, the designers of board games should be output. For these people, the properties "designed by", "author", and "developer" are being used.

Regarding the more complex templates, we found that certain subquery-inducing templates contain each other. These combinations have certain patterns in the NL question corresponding to them.

However, except for **arg_agg_all** containing **agg**, there is not enough data to conclude what they are, and we have to content ourselves with plausible-sounding hypotheses.

7 Conclusion

In this thesis, we identified a comprehensive set of SPARQL templates rooted in semantic aspects of the user’s inquiry. We were able to successfully apply these syntactic building blocks to a benchmark containing 60 examples by generating a template-based query for each example.

The benchmark aims to overcome various shortcomings found in other Wikidata-based benchmarks, such as a lack of complex query structures, and the inclusion of flawed queries that often rely on world knowledge.

Beyond that, the benchmark was created to explore the idea of using Wikipedia lists as a verifiable ground. This approach worked quite well. However, the full range of query structures could only be covered by formulating questions about certain aspects or parts of the Wikipedia lists, rather than just re-creating all their rows and columns. This also had the advantage of having NL questions readily available for analysis purposes.

The templates were made to ensure the correctness and general applicability of the resulting queries, taking into account ties in rankings and including values of zero when taking counts.

With the Wikipedia lists as a "model output", we could uncover structural inconsistencies in Wikidata with ease. By capturing those inconsistencies, the benchmark queries yielded more comprehensive outputs. We also noticed that there is no simple way to return (arguments of) values on certain ranks within groups, as would be needed for examples like "What are the two largest countries on each continent?". This problem became apparent while we were trying to fill this semantic gap or asymmetry by finding a matching template. The easiest way to solve this problem would be to extend the functionality of SPARQL engines.

Since the *Wikipedia Lists* benchmark is fully structured by templates, it is easy to analyze how often each template type occurs in it. Despite only containing 60 examples, the benchmark uses more than 1,000 instances of templates. This is because the re-creation of full Wikipedia lists often requires large queries. To give an idea of what can be done, we performed some preliminary frequency analyses of the different templates, both individually and in combination with others. We analyzed in more depth how certain subquery-inducing templates are nested within each other, and how these combinations relate to patterns in the NL question.

Future work could be directed towards further extending and polishing the benchmark, e.g., by removing

problematic examples and by making it fully usable as a benchmark for testing KGQA systems. To do so, one could use the JSON format employed by the QALD benchmarks. In this file, instead of providing a link to the Wikipedia lists, one could specify the contents of the Wikipedia lists that should be recreated. Based on the semantic templates and categories that were identified, one could also create a graphical user interface to more easily write quality queries, or add the functionality to an existing GUI of a SPARQL engine. While working on the handmade queries of the benchmark, the templates turned out to be a helpful aid during the construction of large, nested queries. Lastly, one could work towards the development of a KGQA system that leverages the semantic templates and categories and combines it with a different, e.g., deep learning-based, approach.

8 Acknowledgements

I would like to express my sincere thanks to my supervisor, Prof. Hannah Bast. She provided me with invaluable feedback whenever I needed it while also giving me the freedom to explore and "play around" with various options. Her challenge to the claim that the problem of KGQA was too complex to be solved with a template-based approach is what inspired me to tackle this topic in the first place.

I am also deeply thankful to my beloved partner, Paul Daum, for his helpful criticism, proofreading and unwavering support, and to my "second family", Nikolas and Monika, for always being there for me.

9 Appendix

Figure 6: Template containment relations without grouping for *Wikipedia Lists*

		Contained template																	
		add_path	path	optional	select	add_name	bind	agg	union	filter	arg_agg_all	minus	agg_all	arg_agg	arg_ranks_all	connect	val_ranks_all	add_desc	
Containing template	add_path	187	74	7		3	4		13	7		6	2	2	1	2			
	union	28	114	3			6		8	4		2						1	
	optional	9	31	47		34	29		5	8					2				
	order				40														
	bind	29	8	7		3	22	6	3					3			1		
	add_name	14	28	14			4	1	4	8	8	2			1	3	1		1
	agg	4	7	5		23	1		4	1		6		1					1
	select					22	8	20		2			5			1		2	
	minus	8	12				1		16								1		
	arg_agg	14								2		2							
	filter	12		3		2	5	6		2	1	2		1					1
	arg_ranks_all	8						4							2				
	connect	3	6										1				2		
	arg_agg_all	1					1	5	1	1									
	agg_all	1	1			1	2			1								1	
	add_desc	1				2													
	val_ranks_all	2						2											
	path																		

Bibliography

- [1] K. Höffner, S. Walter, E. Marx, R. Usbeck, J. Lehmann, and A.-C. Ngonga Ngomo, “Survey on challenges of question answering in the semantic web,” *Semantic Web*, vol. 8, no. 6, p. 895–920, 2017.
- [2] V. Lopez and E. Motta, “Poweraqua: Supporting users in querying and exploring the semantic web content,” *Semantic Web*, vol. 3, no. 3, pp. 249–265, 2012.
- [3] S. He, S. Liu, Y. Chen, G. Zhou, K. Liu, and J. Zhao, “Casia@qald-3: A question answering system over linked data,” in *Working Notes for CLEF 2013 Conference , Valencia, Spain, September 23-26, 2013* (P. Forner, R. Navigli, D. Tufis, and N. Ferro, eds.), vol. 1179 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2013.
- [4] M.-M. Rahoman and R. Ichise, “An automated template selection framework for keyword query over linked data,” vol. 7774 of *Lecture Notes in Computer Science*, pp. 175–190, Springer, 2013.
- [5] H. Bast and E. Haussmann, “More accurate question answering on freebase,” *CIKM ’15: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pp. 1431–1440, 2015.
- [6] S. Shekarpour, E. Marx, A.-C. Ngonga Ngomo, and S. Auer, “Sina: Semantic interpretation of user queries for question answering on interlinked data,” *Journal of Web Semantics*, pp. 39–51, 2015.
- [7] A. Abujabal, M. Yahya, M. Riedewald, and G. Weikum, “Automated template generation for question answering over knowledge graphs,” in *WWW ’17: Proceedings of the 26th International Conference on World Wide Web*, (Republic and Canton of Geneva, CHE), p. 1191–1200, International World Wide Web Conferences Steering Committee, 2017.
- [8] C. Unger, L. Bühmann, J. Lehmann, A.-C. Ngonga Ngomo, D. Gerber, and P. Cimiano, “Template-based question answering over rdf data,” *WWW ’12: Proceedings of the 21st International Conference on World Wide Web*, pp. 639–648, 2012.

- [9] S. Park, H. Shim, and G. G. Lee, “Isoft at qald-4: Semantic similarity-based question answering system over linked data,” in *Working Notes for CLEF 2014 Conference, Sheffield, UK, September 15-18, 2014* (L. Cappellato, N. Ferro, M. Halvey, and W. Kraaij, eds.), vol. 1180 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2014.
- [10] J. Berant and P. Liang, “Semantic parsing via paraphrasing,” *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, vol. 1, pp. 1415–1425, 2014.
- [11] A. Formica, I. Mele, and F. Taglino, “A template-based approach for question answering over knowledge bases,” *Knowledge and Information Systems*, 2023.
- [12] W. Zheng, L. Zou, X. Lian, J. X. Yu, S. Song, and D. Zhao, “How to build templates for rdf question/answering: An uncertain graph similarity join approach,” *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.
- [13] W. Cui, Y. Xiao, H. Wang, Y. Song, S.-w. Hwang, and W. Wang, “Kbqa: Learning question answering over qa corpora and knowledge bases,” *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 565–576, 2017.
- [14] W. Zheng, J. X. Yu, L. Zou, and H. Cheng, “Question answering over knowledge graphs: Question understanding via template decomposition,” *Proc. VLDB Endow.*, vol. 11, no. 11, p. 1373–1386, 2018.
- [15] J. Ding, W. Hu, Q. Xu, and Y. Qu, “Leveraging frequent query substructures to generate formal queries for complex question answering,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 2614–2622, Association for Computational Linguistics, 2019.
- [16] J. J. Gomes, R. Chrispim de Mello, V. Ströele, and J. Francisco de Souza, “A hereditary attentive template-based approach for complex knowledge base question answering systems,” *Expert Systems with Applications*, vol. 205, 2022.
- [17] I. W. Wikimedia Foundation, “Wikidata.” https://www.wikidata.org/wiki/Wikidata:Main_Page. [Online; accessed 15-November-2023].
- [18] D. Vrandečić and M. Krötzsch, “Wikidata: A free collaborative knowledgebase,” *Communications of the ACM*, vol. 57, no. 10, pp. 78–85, 2014.
- [19] I. W. Wikimedia Foundation, “Wikidata Query Service.” <https://query.wikidata.org/>. [Online; accessed 15-November-2023].

- [20] D. Diefenbach, T. P. Tanon, K. Singh, and P. Maret, “Question Answering Benchmarks for Wikidata,” in *ISWC 2017*, 2017.
- [21] D. Sorokin and I. Gurevych, “Modeling semantics with gated graph neural networks for knowledge base question answering,” in *Proceedings of the 27th International Conference on Computational Linguistics*, pp. 3306–3317, Association for Computational Linguistics, 2018.
- [22] A. Saha, V. Pahuja, M. M. Khapra, K. Sankaranarayanan, and S. Chandar, “Complex sequential question answering: Towards learning to converse over linked question answer pairs with a knowledge graph,” 2018.
- [23] M. Dubey, D. Banerjee, A. Abdelkawi, and J. Lehmann, “Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia,” in *International Semantic Web Conference*, pp. 69–78, Springer, 2019.
- [24] S. Cao, J. Shi, L. Pan, L. Nie, Y. Xiang, L. Hou, J. Li, B. He, and H. Zhang, “KQA pro: A dataset with explicit compositional programs for complex question answering over knowledge base,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6101–6119, Association for Computational Linguistics, 2022.
- [25] A. Perevalov, D. Diefenbach, R. Usbeck, and A. Both, “Qald-9-plus: A multilingual dataset for question answering over dbpedia and wikidata translated by native speakers,” 2022.
- [26] R. Usbeck, X. Yan, A. Perevalov, L. Jiang, J. Schulz, A. Kraft, C. Moeller, J. Huang, J. Reineke, A.-C. Ngonga Ngomo, M. Saleem, and A. Both, “QALD-10 - The 10th Challenge on Question Answering over Linked Data.” <https://github.com/KGQA/QALD-10>. [Online; accessed 26-September-2023].
- [27] R. Angles, C. Buil Aranda, A. Hogan, C. Rojas, and D. Vrgoc, “Wdbench: A wikidata graph query benchmark,” in *The Semantic Web - ISWC 2022 - 21st International Semantic Web Conference, Virtual Event, October 23-27, 2022, Proceedings*, vol. 13489 of *Lecture Notes in Computer Science*, pp. 714–731, Springer, 2022.
- [28] M. Krötzsch, J. Gonsior, A. Bielefeldt, L. González, and S. Malyshev, “Wikidata SPARQL Logs.” https://iccl.inf.tu-dresden.de/web/Wikidata_SPARQL_Logs/en. [Online; accessed 10-November-2023].
- [29] C. Davril, “Semantic SPARQL Templates for Question Answering over Wikidata.” <https://ad-blog.cs.uni-freiburg.de/post/semantic-sparql-templates-for-question-answering-over-wikidata/>. [Online; accessed 10-November-2023].

- [30] C. f. A. University of Freiburg and D. Structures, “QLever.” <https://qllever.cs.uni-freiburg.de/wikidata>. [Online; accessed 15-November-2023].
- [31] S. Harris, A. Seaborne, and E. Prud’hommeaux, “SPARQL 1.1 Query Language.” <https://www.w3.org/TR/sparql11-query/>. [Online; accessed 10-November-2023].
- [32] H. Bast and B. Buchhold, “QLever: A query engine for efficient sparql+text search,” in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pp. 647–656, ACM, 2017.
- [33] T. B. of Trustees of The Leland Stanford Junior University, “Stanford Lexicalized Parser v2.0.4 - 2012-11-12.” <https://github.com/chbrown/stanford-parser/blob/master/README.md>. [Online; accessed 11-November-2023].

