



PS 703301 – WS 2021/22
Current Topics in Computer Science

Final Report

Shaping Knowledge Graphs

Philipp Gritsch
Jamie Hochrainer
Kristina Magnussen
Danielle McKenney
Valerian Wintner

supervised by
M.Sc. Elwin Huaman

Contents

1. Introduction	3
2. Related Work	3
3. Approach	4
3.1. Technology Stack	4
3.2. Constructing a Subgraph	4
3.3. Generating Constraints	5
3.4. Validating Constraints	5
3.5. Front-end	6
4. Evaluation	8
4.1. Methodology	8
4.2. Runtime	8
4.3. Correctness	9
4.3.1. ShEx Generation	9
4.3.2. ShEx Validation	10
5. Results	11
6. Future work	11
7. Conclusion	11
References	11
A. Contribution Statements	12
B. Appendix	13
B.1. <i>ShEx</i> outputs	13
B.2. Runtime data tables	13

1. Introduction

With the massive amount of data available on the internet, which is growing every day, a convenient, flexible, and efficient way of storing data becomes more and more important. In addition, concrete objects, abstract ideas as well as connections and relationships between entities have to be represented. This is where *knowledge graphs* become important. *Knowledge graphs* structure data in the form of a graph. This graph can contain types, entities, literals, and relationships. A *knowledge graph* allows for flexible data structures and can make it easier to find and process relevant data. However, the datasets stored in such a graph are often inconsistent and prone to containing errors. Working with such datasets can be greatly facilitated by defining a consistent shape for the data, based on the type of entity it represents. This shaping is done by inferring constraints over the data and validating all nodes in the graph based on these constraints. This can give important insight into the structure of the data. For instance, when all nodes of a type conform to the given constraints, it may be useful to define these as required attributes for all future nodes to ensure uniformity in the data. Non conforming nodes may also deliver important insight into where information is missing. For example, if 99% of nodes of a given type conform to some constraints, it may be worthwhile to investigate the remaining 1% to see if they are missing necessary information or are otherwise corrupt.

For this reason, we implemented a framework for shaping *knowledge graphs*. This consisted of three major steps: fetching a subset dataset from a *knowledge graphs*, inferring constraints, and validating a *knowledge graph* against these constraints. We also provide a user interface to this purpose. These steps are described in Section 3. After this was done, we evaluated our framework concerning runtime and correctness which is outlined in Section 4. Results of our project are shown in Section 5. A future outlook is given in Section 6. Finally, a conclusion of our work is provided in Section 7.

2. Related Work

The need for automatic tools that are able to infer meta information on the structure of *knowledge graphs* has already been recognized by different researchers. This stems from the fact that manual constraint inference becomes infeasible for large datasets.

One tool which can be used to automatically infer constraints over a *knowledge graph* is *RDF2Graph*[8, 7]. Our framework makes use of an adapted version of this tool by Werkmeister [9, 10], which uses several *SPARQL* queries to gather the structural information of each node in the underlying graph in a first phase. Subsequently, the queried information is gathered and simplified. This is achieved by merging constraint information of classes belonging to the same type and predicates. While Van Dam et al. used the *RDF2Graph* tool on the *UniPort RDF* resource, Werkmeister made adaptations to also infer Wikidata constraints.

Fernandez-Álvarez et al. have taken a different approach with their tool *Shexer*[3]. In contrast to the aforementioned tool, they avoid querying the whole underlying graph by using an iterative approach, determining whether or not the current iterated (sub-)set of triples is relevant for the constraint generation process. Given a target shape, the preselected triples are used to decorate each target instance with its constraints.

Another constraint generator has been introduced by Spahiu et al. with *ABSTAT*[6]. This tool uses an approach similar to that of *RDF2Graph* by collecting structural information using *SPARQL* queries and summarizing those constraints afterwards.

3. Approach

To construct a framework that offers a way to evaluate a *knowledge graph* in an automated way, we divided our project into three main subtasks. At first, we fetch a subgraph of a *knowledge graph* from the *CommonCrawl* datasets, as explained in greater detail in Section 3.2. After this, our framework infers constraints over this data set (see Section 3.3). In the last step, the subgraph is validated against the constraints (see Section 3.4). The structure of the framework can be seen in Fig. 1. The user can interact with this framework over the front-end (see Section 3.5).

The framework can be found in our git repository [5]. The repository also includes a README file describing how to set-up and install the project.

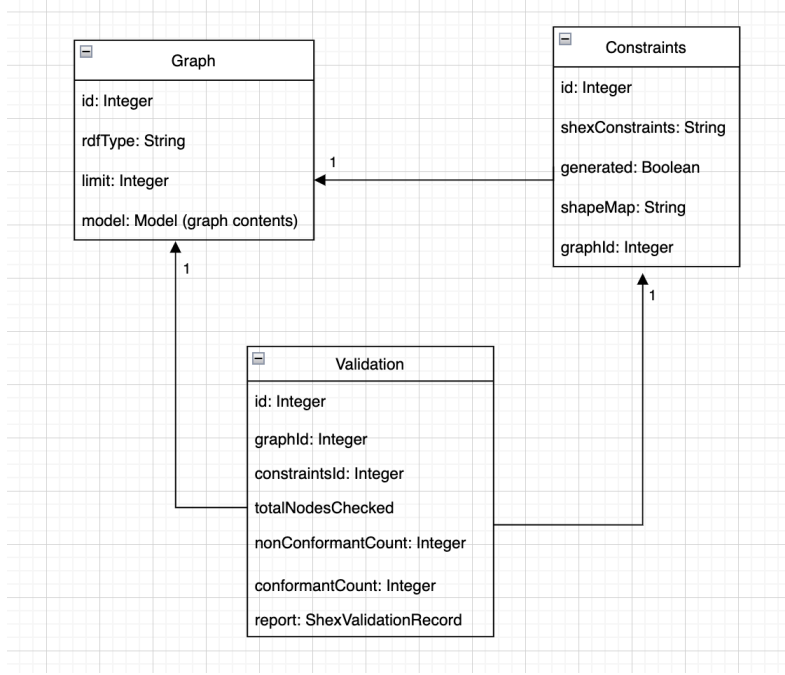


Figure 1: UML diagram of the framework structure

3.1. Technology Stack

In this section, we briefly enumerate the main technologies that we used in this project. We used *Maven* as a project management tool. The framework was implemented in *Java*. Here, we also used the *Java* framework *Jena*[1], which offers an *RDF* API as well as support for *SPARQL* queries and the *ShEx* language. The front-end was implemented using *Vue3*[2] as a front-end framework and *PrimeVue* as a library for the different UI components. For the deployment of our application we used a single virtual machine. Access to the front-end is done via a single *Apache* server. The front-end accesses the back-end via an internal *REST-API*.

3.2. Constructing a Subgraph

Because *knowledge graphs* can be very large and contain many nodes, we concentrated on querying smaller subgraphs and only working on those. With this method, the relevant subgraph gets extracted from a *knowledge graph* and can be worked upon in isolation. We take our initial *knowledge graphs* from the *CommonCrawl* datasets and import them as a static file.

```

1 CONSTRUCT {
2   ?s ?p ?o
3 }
4 %s                                     # %s replaced by optional graph-name
5 WHERE {
6   GRAPH ?g {
7     ?subject (<>|!<>)* ?s .
8     ?s ?p ?o .
9     {
10      SELECT DISTINCT ?subject
11      WHERE {
12        GRAPH ?g { ?subject a %s }    # %s replaced by type of starting-nodes
13      } %s                            # %s replaced by optional limit
14    }
15  }
16 }

```

Figure 2: The *SPARQL*-query creating the subgraph. The *%s* get substituted before executing the query.

Figure 2 shows the query we used to create a subgraph. At line 7 we used *property paths*¹ to query all nodes connected to those of an initial subset (lines 10 to 13). This subset can optionally be limited to a certain size, but is always limited to nodes of a certain type.

3.3. Generating Constraints

To shape a *knowledge graph* we need to infer constraints on the previously fetched subgraph. For the generation of constraints, we used the adaption of the tool *RDF2Graph*[7] by Werkmeister[10] and adapted it for our purposes. As input, *RDF2Graph* takes a constructed subgraph as described in Section 3.2. The properties of the graph are read out with several *SPARQL* queries. These properties are saved in a new *RDF* graph. As output, we receive a graph containing constraints for the initial input data. We use a tool offered by *RDF2Graph* to extract the constraints in *ShEx* syntax.

We implemented the following steps in order to integrate *RDF2Graph* into our project. We added *RDF2Graph* to our framework so that they could be compiled together, and in the process updated it as much as was needed to be compatible with our version of Java and Jena. In addition, we changed some of the initial parameters of the *RDF2Graph*, since it originally was intended as a stand-alone application. As we are handling *Models* in our software, we changed the input from a *RDF2Graph* to a *Model*. In our application, *RDF2Graph* does not use any other storage apart from the *Model* data structure. Previously, such a *Model* needed to be created by *RDF2Graph*; now it is provided by our framework. We did this so we could have full control over the files handled by *RDF2Graph*. *RDF2Graph* allows for multithreaded execution, which requires a thread pool. This thread pool was initially created by *RDF2Graph*. In our framework, it is provided by our application. In addition, resources which are used by *RDF2Graph* had to be provided in a different way so that they are still available when running from a server environment. We also changed some of the queries. *RDF2Graph* supports multiple output graphs, however, this did not work. As we only work on one *Model* at a time, we only use one output graph.

3.4. Validating Constraints

Given a *RDF* graph and a set of constraints, the validation consists of verifying that every node in the graph fulfils the requirements given in the constraints. A graph may contain node with different types. Each of those types must conform to its corresponding definition outlined in the constraints. The result of the validation is a multidimensional list containing every node's id, a boolean flag, and an optional 'reason' entry. The boolean flag indicates whether or not the node conforms to its type's constraints. In case of nonconformity, a reason will be given.

¹<https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#propertypaths>

```

1 SELECT DISTINCT ?type
2 WHERE {
3   ?s a ?type
4 }

```

Figure 3: The very simple query getting the different types to be used in the *shape map*.

```

1 {FOCUS a <https://schema.org/GeoCoordinates>@<https://schema.org/GeoCoordinates>,
2 {FOCUS a <https://schema.org/RiverBodyOfWater>@<https://schema.org/RiverBodyOfWater>,
3 {FOCUS a <https://schema.org/Service>@<https://schema.org/Service>,
4 {FOCUS a <https://schema.org/Canal>@<https://schema.org/Canal>,
5 {FOCUS a <https://schema.org/AdministrativeArea>@<https://schema.org/AdministrativeArea>,
6 {FOCUS a <https://schema.org/Map>@<https://schema.org/Map>,
7 {FOCUS a <https://schema.org/Place>@<https://schema.org/Place>,
8 {FOCUS a <https://schema.org/EducationalOrganization>@<https://schema.org/
   EducationalOrganization>

```

Figure 4: The shape map used for validating the full subgraph starting with RiverBodyOfWater-nodes.

For the implementation of this process, an *RDF* subgraph and *ShEx* constraints are required as input. Then, we use this to generate a *shape map*, which contains all of the types that need to be validated. For the actual validation, the *ShExValidator* provided by the *Jena* library was used. The validator requires a set of constraints defined in valid *ShEx* syntax and a *shape map*. We query the subgraph for its types of nodes (see Figure 3), and construct the *shape map* from that. Figure 4 shows an example.

The class *ShExValidationRecord* stores the result of the validation for each node of the graph. Additionally, the percentage of nodes that conform to their constraints is calculated and stored.

3.5. Front-end

We implemented a front-end where the user can choose a *knowledge graph* as well as its type (see Figure 5). In addition, the user can also set a limit on the number of nodes of the specified type that they wish to have constraints generated for. As output (see Figure 6), *ShEx* constraints as well as a validation of the subgraph against those constraints are returned. The constraints can be edited by the user and the selected subgraph can be re-validated against the newly edited constraints. If a node is deemed invalid, a reason is given, e.g. "Cardinality violation (min=1): 0". The user can download the subgraph that was validated. The interaction between user, front-end and server can also be seen in Fig. 7.

[Home](#) | [Edit](#)

Validate RDF data with ShEx

Select a **Dataset**:

RiverBodyOfWater ▼

Select a **Type**:

RiverBodyOfWater ▼

Specify a **Limit** (No Limit = 0):

200

+

Calculate Constraints

Figure 5: The frontend, showing a selection of dataset, *RDFType*, and *LIMIT* of starting-nodes.

Validate RDF data with ShEx

Constraints for **RiverBodyOfWater/RiverBodyOfWater** with **limit 200** . [Download](#) ↓

[Go back to constraint selection](#)

Shapes graph (ShEx) input:

```

1 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
5 PREFIX schema: <http://schema.org/>
6
7 <https://schema.org/AdministrativeArea> {
8   <https://schema.org/name> rdf:langString;
9   <https://schema.org/url> .
10 }
11
12 <https://schema.org/Canal> {
13   <https://schema.org/description> rdf:langString?;
14   <https://schema.org/name> rdf:langString?;

```

Revalidate

✓ Accept valid ShEx

Validation result:

Conformant: 414 / 414

Focus	Status ↑↓	Reason
b75f9370da9a3b19f1694599f863be0a	conformant	
e858f86a7a156001df37b8a5925bae6d	conformant	
bb2a5c9714113fd297059736041f1833	conformant	
e2b7f425550a0c483935281440be0aba	conformant	
f27b7b11b30531ba57de0df477b863f4	conformant	
0b6eb4e0dcbaf9ca45c2f36dad4502d	conformant	
69261a4df78e5f53d8892d5af92466d7	conformant	
5bebce0b7bd2df9172528379e4c13d7d	conformant	
eb780280575fd75caf9cc2cbf58d08cf	conformant	
6489bfdc8783b7b1d55a23b4feda35ff	conformant	

(1 of 42) << < 1 2 3 4 5 > >> 10 ▾

Figure 6: The frontend, showing the calculated *ShEx*-constraints and validation-results.

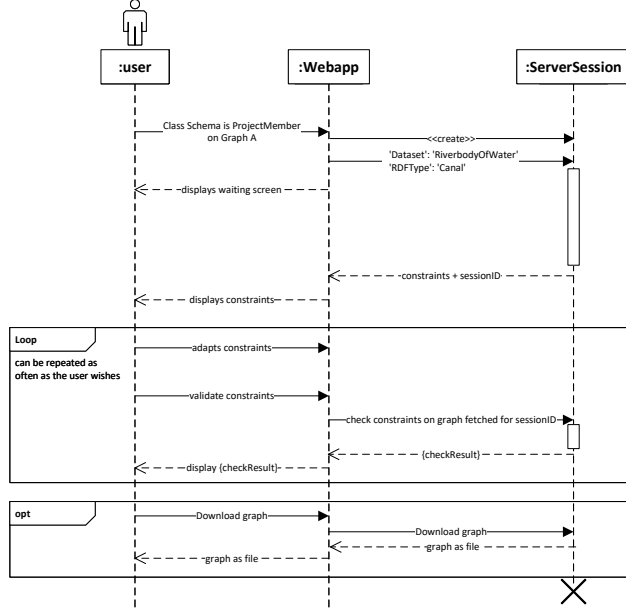


Figure 7: Sequence diagram showing the interaction between web application, user and server

4. Evaluation

In this section we evaluate our tool. We explain the methodology in Section 4.1. In Section 4.2 we measured the runtime of our tool with different input parameters. Furthermore, we tested correctness of the generated *ShEx*-constraints and also cross validated them in Section 4.3.

4.1. Methodology

For taking measurements, the application was started locally on our hardware. This was done to minimise side-effects of other applications running on the virtual machine where the live-instance is deployed. We used a machine with a *Ryzen 9 3900x* CPU with 12x3.8GHz cores, DDR4 RAM and an SSD. Additionally, the JVM was set up to use up to 16 GB of main memory for its heap to allow parallel queries without compromising the runtime of the executions, arising from extensive swap usage.

4.2. Runtime

Figures 8 and 9 show the measurements we obtained by changing the *LIMIT* input parameter. This parameter limits the size of the start-node subset, from which connected nodes are queried. All the measurements are shown in Tables 2 and 1.

The results shown in Figure 8 were to be expected. First of all, the runtime of constructing the desired subset of the graph is considerably larger than the time needed to create the *ShEx* constraints, or to validate the constraints on the graph. Secondly, the runtime of constructing the subgraph scales with the *LIMIT*. This becomes especially evident in Figures 8a and 8c.

To understand the behaviour shown in Figure 8b, we want to look at Figure 9, which shows the same runtimes, but grouped by the number of triples in the subgraph on which the constraints are created. As opposed to Figures 9a and 9c, the maximum number of triples (shown in the x-coordinate in Figure 9b), is 1769. This is also the amount of triples contained in the subgraph that we get without providing any

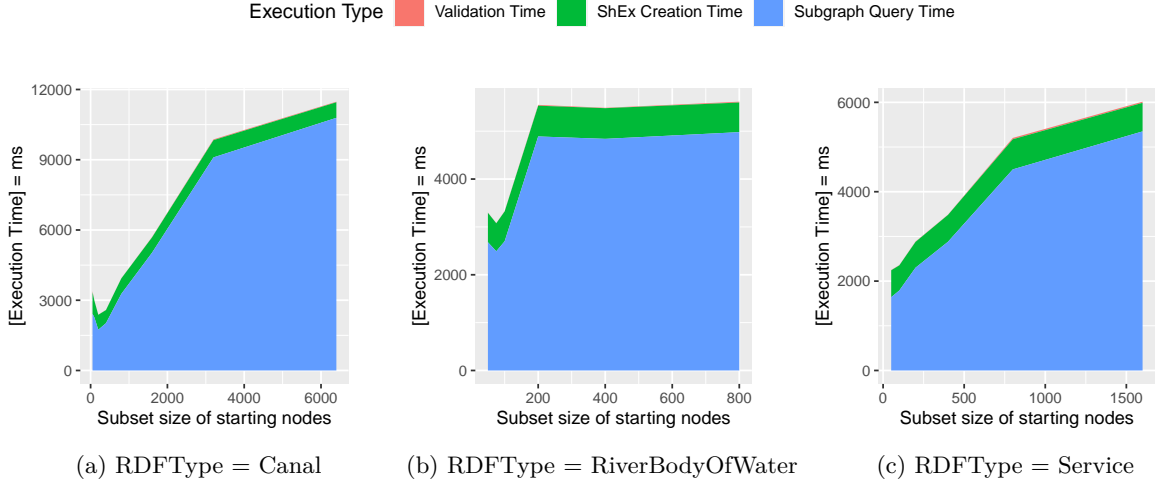


Figure 8: Execution times per RDFType, per size of start-node subset on RiverBodyOfWater dataset

limit. Therefore, providing a limit larger than 200 won't enrich the constructed graph, keeping the time almost constant in regards to the *LIMIT* parameter.

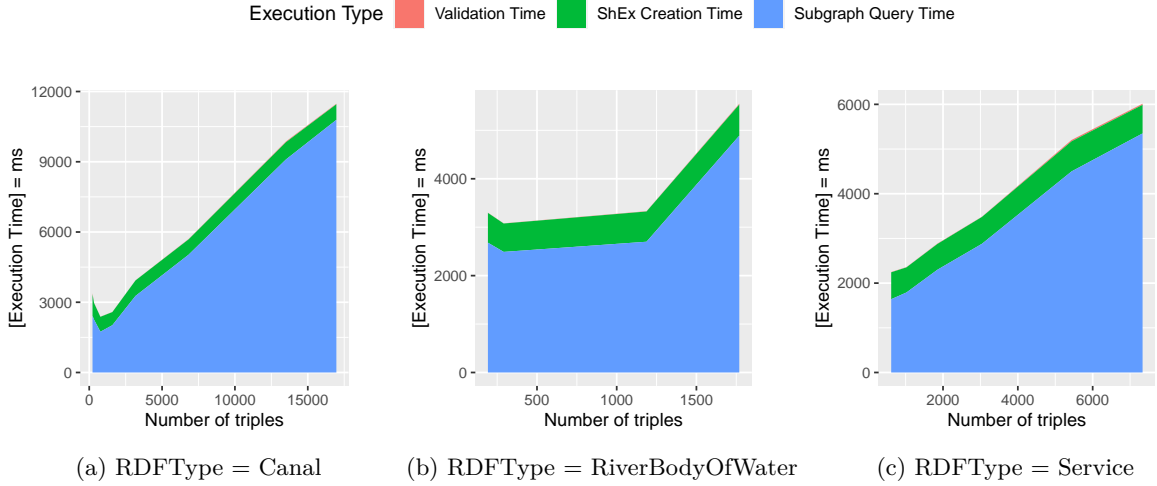


Figure 9: Execution times per RDFType, per number of triples on RiverBodyOfWater dataset

Figure 10 shows the runtime without limiting the construction of the subgraph. Note the much larger runtime needed for querying the graph, despite resulting in the same amount of triples when providing a large enough *LIMIT*.

4.3. Correctness

4.3.1. ShEx Generation

We thought *Shexer* (see Section 2) was a good fit for cross validating our *ShEx*-generation. However, due to our limited knowledge of operating this tool, we did not manage to generate proper constraints for

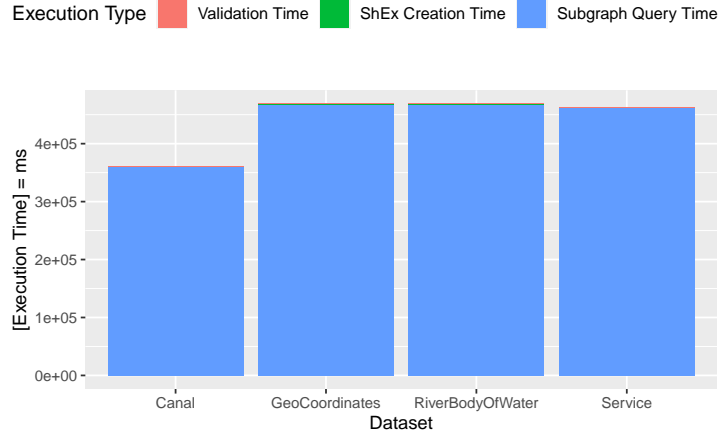


Figure 10: Execution times per RDFType of the RiverBodyOfWater dataset (containing 49915 triples)

```

1 [ a      <https://schema.org/Service> ;
2   <https://schema.org/serviceType>
3     "Marine Electrics"@en
4 ] .
5
6 [ a      <https://schema.org/Service> ;
7   <https://schema.org/description>
8     "A list of locations with dry dock facilities on the Main Canal of the Trent &
9     Mersey Canal"@en ;
10  <https://schema.org/serviceType>
11    "Dry Dock"@en ;
12  <https://schema.org/url> <https://www.ukwaterwaysguide.co.uk/s/trent-mersey-canal/
    main-canal/dry-dock>
13 ] .

```

Figure 11: Blank Nodes in Turtle File

our RiverBodyOfWater-dataset. Our attempt at using this tool is shown in Figure 12, which generated the trivial, non-restrictive constraints shown in Figure 13.

Therefore, we checked the generated constraints manually for small subgraphs (see Figures 14, 15 and 16) and identified two issues with our tool.

Firstly, if the dataset consists of only stand-alone blank nodes, as seen in Figure 11, then *Rdf2Graph* does not infer any *ShEx*-constraints. This was the case for the generated subgraph using RiverBodyOfWater with a *LIMIT* of 50, and the resulting *ShEx* can be seen in Figure 15.

Secondly, optional properties are not always inferred and therefore missing from the generated *ShEx*-constraints. This also happens for unlimited subgraphs (see Figures 17, 18 and 19), with the exception of the RiverBodyOfWater-RDFTYPE, where it looks like the constraints are complete. However, due to the size of the graph manually checking for correctness is infeasible. We did not see a correlation between missing constraint-properties and the shape of the graph.

4.3.2. ShEx Validation

The generated *ShEx*-constraints for small subgraphs (Canal with *LIMIT* 50, Service with *LIMIT* 50) were cross validated using the online-tool *RDFShape*[4]. The validation result was the same as in our tool.

5. Results

Our framework automatically infers constraints and validates the given data based on those constraints. This can be done on two different *CommonCrawl* datasets. The user can choose one of those datasets and a limit using the front-end. In addition, the user can also edit the constraints. Our evaluation in Section 4 showed that the validation of a subgraph works as expected. However, the constraints generated are prone to missing an optional *url* attribute and would benefit from more tests and tuning. In addition, we could also see that the runtime of the tool is rather slow. Possible improvements to our framework are discussed in more depth in Section 6. Generally, our tool works as intended, even though there is still some room for improvement.

6. Future work

Our application currently only handles two different datasets. For future work, this could be expanded so that the framework could handle more and bigger datasets. Currently, the size of the datasets that can be handled is limited by the RAM on the virtual machine. One possible solution for this could be to only work on parts of the graph. One problem we encountered when handling datasets from *CommonCrawl* was the quality of these datasets. Many datasets include *non-unicode* characters, which are replaced by Jena with *unicode* characters. This takes a lot of computing time. In addition, many files include invalid *RDF* syntax or are otherwise damaged. This means that in order to handle additional datasets, some way of processing these datasets would have to be implemented. Processing could include filtering for broken files and invalid syntax and fixing this before handling the dataset in the framework. In addition, more possibilities for user interaction could be added. For instance, a feature could be added where a user can upload their own dataset and have it validated.

7. Conclusion

Overall, we achieved the creation of a functional interface that allows a user to view and edit automatically generated constraints for a given graph. A highlight of our tool certainly is its ease of use, simply presenting a user with a drop-down list of *RDF* types that they can evaluate. Although the selection is minuscule at the moment, this could be trivially expanded to allow a greater selection of *RDF* types from the *CommonCrawl* dataset. One of the persistent flaws of the tool we developed remains its poor performance with larger graphs, however, to a certain extent, this is inevitable when working with such large amounts of data. Such a complex *SPARQL* query that fetches an 'infinite depth' of related objects is bound to have a relatively slow runtime. In addition, although the vast majority of the constraints that could be present on a graph do get generated, additional, as well as automated testing would be required to increase confidence in the completeness and correctness of said constraints. The validation of these constraints though, works as expected, and without performance issues. In the end, we succeeded in developing a working prototype that could form the base of a more powerful, flexible tool for easily gaining insight into any *knowledge graph*.

References

- [1] Apache Jena, documentation. <https://jena.apache.org/index.html>.
- [2] Vue.js, documentation. <https://v3.vuejs.org/>.
- [3] D. Fernandez-Álvarez, J. E. Labra-Gayo, and D. Gayo-Avello. Automatic extraction of shapes using shexer. *Knowledge-Based Systems*, 238:107975, 2022.

- [4] J. E. L. Gayo. Rdfshape. <https://rdfshape.herokuapp.com/validate>, 2021. Accessed: 2022-01-23.
- [5] D. McKenney, K. Magnussen, P. Gritsch, V. Wintner, and J. Hochrainer. Kg shapes. <https://git.uibk.ac.at/csaz8448/kg-shapes>, 2022. Accessed: 2022-02-04.
- [6] B. Spahiu, R. Porrini, M. Palmonari, A. Rula, and A. Maurino. Abstat: Ontology-driven linked data summaries with pattern minimalization. In *SumPre@ESWC*, 2016.
- [7] J. van Dam. Rdf2graph. <https://github.com/jessevdam/RDF2Graph/>, 2022. Accessed: 2022-01-16.
- [8] J. C. van Dam, J. J. Koehorst, P. J. Schaap, V. A. Martins dos Santos, and M. Suarez-Diez. Rdf2graph a tool to recover, understand and validate the ontology of an rdf resource. *Journal of Biomedical Semantics*, 6(1):39, Oct 2015.
- [9] L. Werkmeister. Schema inference on wikidata. Master’s thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, 2018.
- [10] L. Werkmeister. Rdf2graph. <https://github.com/lucaswerkmeister/RDF2Graph>, 2022. Accessed: 2022-01-16.

A. Contribution Statements

Each member of the group contributed in an enthusiastic and equal manner, leveraging their individual skills to contribute to the parts of the project where they could make the most impact. Additionally, everyone was eager to learn new skills and teach others what they knew. We are all satisfied with how much everyone contributed and how well we worked together. The following presents a brief summary of each group member’s contribution to the project:

- Danielle’s programming and organizational skills were a great asset to the team. She ensured that meetings had structure, with clear goals, responsibilities, and deadlines defined. She worked most the implementing the *ShEx* validation and developing the backend app, leading several pair programming sessions with her team members. She also participated in the final presentation and parts of the report.
- Jamie contributed most with her teamwork skills and adaptability. Although she did not have as much experience programming in java and javascript as other team members, she readily made an effort to learn and thrived with the pair programming method we implemented, working largely on the webapp and parts of the *RDF2Graph* implementation. Additionally, she was heavily involved in the designing the presentations and reviewing the report.
- Kristina contributed most in the research and planning of the project. Her research skills were heavily utilized in the initial phase of the project, which greatly helped the others when it came to choosing libraries and overcoming difficulties in the implementation of technical problems. She worked most on programming parts of the project requiring knowledge of *ShEx*. Her extra research also proved useful in delivering thorough and well thought out presentations and drafting the report.
- Philipp’s technical skills were highly useful in the programming part of the project. He advised the selection of the tech stack and led many pair programming sessions, readily sharing his technical knowledge with the other team members. This also came in handy when contributing the evaluation and ‘related work’ sections of the report.
- Valerian, similarly to Philipp, also had strong technical skills that he applied in various areas of the project. He worked on the *SPARQL* parts of the programming and on creating the frontend app, often leading a pair programming session. He also contributed to the evaluation and writing up the results of this.

```

1 from shexer.shaper import Shaper
2 from shexer.consts import NT, SHEXC
3
4 namespaces_dict = {
5     "http://www.w3.org/2001/XMLSchema#": "xsd",
6     "http://www.w3.org/1999/02/22-rdf-syntax-ns#": "rdf",
7     "http://www.w3.org/2000/01/rdf-schema#": "rdfs",
8     "http://www.w3.org/2004/02/skos/core#": "skos",
9     "http://schema.org/": "schema"
10 }
11
12 input_file = "rbow.nt"
13
14 shaper = Shaper(
15     graph_file_input=input_file,
16     all_classes_mode=True,
17     input_format=NT,
18     remove_empty_shapes=False,
19     discard_useless_constraints_with_positive_closure=False,
20     depth_for_building_subgraph=100,
21     inverse_paths=True,
22     shapes_namespace="http://schema.org/",
23     all_instances_are_compliant_mode=False,
24     namespaces_dict=namespaces_dict,
25     instantiation_property="http://www.w3.org/1999/02/22-rdf-syntax-ns#type") # Default
26     rdf:type
27
28 output_file = "shexer_rbow.shex"
29
30 shaper.shex_graph(output_file=output_file,
31                   output_format=SHEXC,
32                   acceptance_threshold=.1)

```

Figure 12: Running shexer on the full graph

B. Appendix

This appendix shows *ShEx*-output and tables of data that were too large to be in the main body. Section 4 links to some of the figures listed here.

B.1. *ShEx* outputs

- Figure 13 shows the *Shexer*-output from our attempt at using it. The code is shown in Figure 12.
- Figures 14, 15 and 16 show our generated *ShEx* for subgraphs with a small *LIMIT*, sometimes resulting in graphs only consisting of blank nodes, and therefore empty *ShEx* constraints.
- Figures 17, 18 and 19 show our generated *ShEx* for subgraphs without any *LIMIT*.

B.2. Runtime data tables

- Table 1 shows the runtimes with different *LIMIT*s given.
- Table 2 shows the runtimes without any *LIMIT* given.

```

1 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
5 PREFIX : <http://schema.org/>
6
7 :PropertyValue
8 {
9 }
10
11
12 :RiverBodyOfWater
13 {
14 }
15
16
17 :Hotel
18 {
19 }
20
21 # ... many more empty terms

```

Figure 13: Shexer output

```

1 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
5 PREFIX schema: <http://schema.org/>
6
7 <https://schema.org/Canal> {
8   <https://schema.org/description> rdf:langString?;
9   <https://schema.org/name> rdf:langString?;
10  #<https://schema.org/url> .? # <-- missing from generated ShEx.
11 }
12
13 <https://schema.org/RiverBodyOfWater> {
14   <https://schema.org/description> rdf:langString;
15   <https://schema.org/name> rdf:langString;
16   <https://schema.org/url> .
17 }
18
19 <https://schema.org/Service> {
20   <https://schema.org/description> rdf:langString?;
21   <https://schema.org/serviceType> rdf:langString;
22   <https://schema.org/url> .?
23 }

```

Figure 14: Generated *ShEx*-constraints of Canal with *LIMIT* 50

```

1 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
5 PREFIX schema: <http://schema.org/>

```

Figure 15: Generated *ShEx*-constraints of RiverBodyOfWater with *LIMIT* 50

```

1 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
5 PREFIX schema: <http://schema.org/>
6
7 <https://schema.org/Canal> {
8   <https://schema.org/description> rdf:langString?;
9   <https://schema.org/name> rdf:langString?;
10  #<https://schema.org/url> .? # <-- missing from generated ShEx.
11 }
12
13 <https://schema.org/RiverBodyOfWater> {
14   <https://schema.org/description> rdf:langString;
15   <https://schema.org/name> rdf:langString;
16   <https://schema.org/url> .
17 }
18
19 <https://schema.org/Service> {
20   <https://schema.org/description> rdf:langString?;
21   <https://schema.org/serviceType> rdf:langString;
22   #<https://schema.org/url> .? # <-- missing from generated ShEx.
23 }

```

Figure 16: Generated *ShEx*-constraints of Service with *LIMIT* 50

```

1 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
5 PREFIX schema: <http://schema.org/>
6
7 <https://schema.org/Canal> {
8   <https://schema.org/address> rdf:langString?; # <-- Only 1 standalone blank node with
9     this property
10   <https://schema.org/description> rdf:langString?;
11   <https://schema.org/name> rdf:langString?;
12   #<https://schema.org/url> .? # <-- missing from generated ShEx.
13 }
14
15 <https://schema.org/RiverBodyOfWater> {
16   <https://schema.org/description> rdf:langString;
17   <https://schema.org/name> rdf:langString;
18   <https://schema.org/url> .
19 }
20
21 <https://schema.org/Service> {
22   <https://schema.org/description> rdf:langString?;
23   <https://schema.org/serviceType> rdf:langString;
24   #<https://schema.org/url> .? # <-- missing from generated ShEx.

```

Figure 17: Generated *ShEx*-constraints of Canal without a *LIMIT*

```

1 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
5 PREFIX schema: <http://schema.org/>
6
7 <https://schema.org/AdministrativeArea> {
8   <https://schema.org/name> rdf:langString;
9   <https://schema.org/url> .
10 }
11
12 <https://schema.org/Canal> {
13   <https://schema.org/description> rdf:langString?;
14   <https://schema.org/name> rdf:langString?;
15   <https://schema.org/url> .?
16 }
17
18 <https://schema.org/EducationalOrganization> {
19   <https://schema.org/name> rdf:langString;
20   <https://schema.org/url> .
21 }
22
23 <https://schema.org/GeoCoordinates> {
24   <https://schema.org/elevation> rdf:langString?;
25   <https://schema.org/latitude> rdf:langString?;
26   <https://schema.org/longitude> rdf:langString?
27 }
28
29 <https://schema.org/Map> {
30   <https://schema.org/sameAs> .
31 }
32
33 <https://schema.org/Place> {
34   (
35     <https://schema.org/containsPlace> @<https://schema.org/AdministrativeArea>* |
36     <https://schema.org/containsPlace> @<https://schema.org/EducationalOrganization>?
37   );
38   <https://schema.org/name> rdf:langString?;
39   (
40     <https://schema.org/url> .? |
41     <https://schema.org/url> rdf:langString?
42   )
43 }
44
45 <https://schema.org/RiverBodyOfWater> {
46   <https://schema.org/address> rdf:langString?;
47   <https://schema.org/alternateName> rdf:langString*;
48   <https://schema.org/containedInPlace> @<https://schema.org/Place>*;
49   <https://schema.org/description> rdf:langString*;
50   <https://schema.org/geo> @<https://schema.org/GeoCoordinates>*;
51   <https://schema.org/hasMap> @<https://schema.org/Map>*;
52   <https://schema.org/image> rdf:langString*;
53   <https://schema.org/name> rdf:langString+;
54   <https://schema.org/sameAs> .*
55 }
56
57 <https://schema.org/Service> {
58   <https://schema.org/description> rdf:langString?;
59   <https://schema.org/serviceType> rdf:langString;
60   <https://schema.org/url> .?
61 }

```

Figure 18: Generated *ShEx*-constraints of *RiverBodyOfWater* without a *LIMIT*


```

1 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
5 PREFIX schema: <http://schema.org/>
6
7 <https://schema.org/Canal> {
8   <https://schema.org/description> rdf:langString?;
9   <https://schema.org/name> rdf:langString?;
10  #<https://schema.org/url> .? <-- Missing from generated ShEx
11 }
12
13 <https://schema.org/RiverBodyOfWater> {
14   <https://schema.org/description> rdf:langString;
15   <https://schema.org/name> rdf:langString;
16   <https://schema.org/url> .
17 }
18
19 <https://schema.org/Service> {
20   <https://schema.org/description> rdf:langString?;
21   <https://schema.org/serviceType> rdf:langString;
22   #<https://schema.org/url> .? # <-- Missing from generated ShEx
23 }

```

Figure 19: Generated *ShEx*-constraints of Service without a *LIMIT*

Rdftype	Limit	Triples	$[t_{graph}] = \text{ms}$	$[t_{shex}] = \text{ms}$	$[t_{validation}] = \text{ms}$
Canal	50	226	2420	923	44
Canal	100	328	2260	709	13
Canal	200	765	1740	637	8
Canal	400	1588	2020	559	9
Canal	800	3176	3270	661	8
Canal	1600	6817	5030	654	17
Canal	3200	13504	9100	736	33
Canal	6400	16961	10790	665	25
RiverBodyOfWater	50	192	2680	615	5
RiverBodyOfWater	75	291	2490	586	4
RiverBodyOfWater	100	1187	2700	624	7
RiverBodyOfWater	200	1769	4890	643	17
RiverBodyOfWater	400	1769	4840	641	8
RiverBodyOfWater	800	1769	4980	619	18
Service	50	615	1640	602	7
Service	100	1022	1790	562	6
Service	200	1852	2300	577	9
Service	400	3041	2880	601	6
Service	800	5437	4500	674	30
Service	1600	7334	5350	639	26

Table 1: Execution times per RDF-Type, limited size of start-node subset (using the RiverBodyOfWater dataset)

Rdftype	Triples	$[t_{graph}] = \text{ms}$	$[t_{shex}] = \text{ms}$	$[t_{validation}] = \text{ms}$
Canal	16961	360000	737	45
GeoCoordinates	204	468000	585	4
RiverBodyOfWater	1769	468000	613	15
Service	7334	462000	618	19

Table 2: Execution times per RDF-Type, queried on full graph of the RiverBodyOfWater dataset (containing 49915 triples)