



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	3
3.1. Technology Stack	3
3.2. Generating Constraints	3
3.2.1. Integrating RDF2Graph with our framework	4
3.3. Validating Constraints	5
3.4. Front-end	5
4. Results	6
4.1. Future work	6
5. Evaluation	7
5.1. Methodology	7
5.2. Runtime	7
5.3. Correctness	9
5.4. ShEx Validation	9
6. Conclusion	9
References	9
A. Contribution Statements	9
B. Appendix	9

I added a comment colour for everyone. ;Comment colour Jamie; ;Comment colour Danielle;
;Comment colour Philipp; ;Comment colour Valerian; ;Comment colour Kristina;

remove this
part

1. Introduction

We used *CommonCrawl* datasets as the base for the *knowledge graph* which we wanted to assess. The data contained in those datasets is often inconsistent and might contain errors. In order to work with this data properly, it is necessary to shape the *knowledge graph* in which this data is contained. This shaping is done by inferring constraints over the data and validating it based on these constraints. Validating a graph against constraints gives important insight into the structure of the data. For instance, when all nodes of a type conform to constraints, then 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 otherwise corrupt.

add introduction to KGs

2. Related Work

3. Approach

Our framework offers a way to evaluate a *knowledge graph* in an automated way. For this, we used *knowledge graphs* from the *CommonCrawl* datasets as a basis. The *knowledge graphs* are imported as a static file. After this, our framework infers constraints over this data set (see Section 3.2). These are validated automatically in the last step, see Section 3.3. The user can interact with this framework over the front-end, see Section 3.4. These different steps were implemented and tested separately. Once this was done, we consolidated them. The structure of our project can be seen in Fig. 1.

Add thesis
Werkmeister
RDF2Graph,
also add another work,
maybe from sources in thesis, done by Philipp

3.1. Technology Stack

The framework was implemented in *Java*. We used *Maven* as a project management tool. We also used *Jena*, which offers an *RDF* API as well as support for *SPARQL* queries and the *ShEx* language. The front-end was implemented using *Vue3*[1] as a front-end framework and *PrimeVue* as a library for the different components. For the deployment of our application we use 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*.

update figure

refer to the
readme here?
Or should this
happen somewhere else?

add reference
to our github
repo!

3.2. Generating Constraints

For the generation of constraints, we used the tool *RDF2Graph* [2] and adapted it for our purposes. As input, *RDF2Graph* takes a *knowledge graph* from *CommonCrawl*. 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 *RDF2Graph* queries to extract the constraints in *ShEx* syntax.

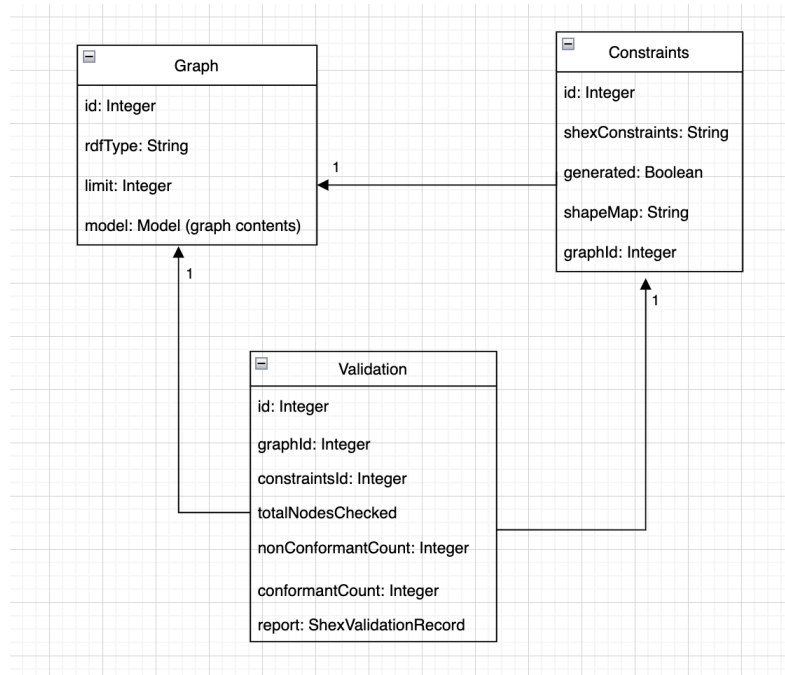


Figure 1: UML diagram of the framework structure



add query to graph (chosen by Philipp), e.g. multiplicity of argument etc.

3.2.1. Integrating RDF2Graph with our framework

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. 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 to *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.

Add explanation for *Model*? May in glossary?

should we explain this in more detail?

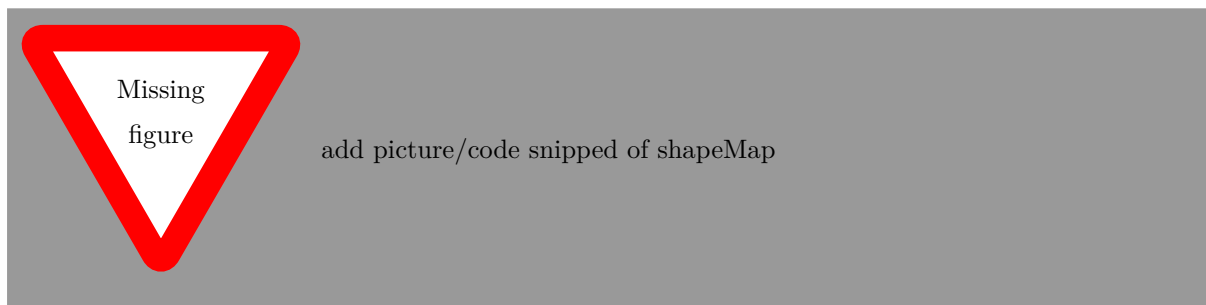
Add explanation of limit this section?

3.3. 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 consists of several different types. Each of those types must conform to its definition outlined in the constraints. The results of the validation is be a boolean flag for every single node in the graph, indicating whether or not it conforms to its type's constraints. In case of nonconformity, a reason will be given.

In our code, this is implemented in the following way. As input, we receive a *RDF* subgraph as well as a set of constraints. We use this to generate a *shape map*, which contains all of the types which 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*. The *shape map* describes which types of nodes need to be validated against which *ShEx* constraint definitions.

add reference
to Jena libra
here?

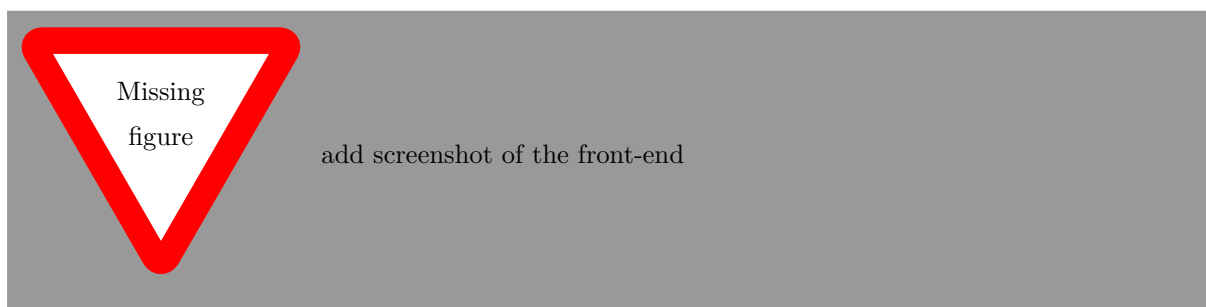


The class *ShexValidationRecord* stores the result of the validation for every single node of the graph. Not only is the individual result of every node checked against its relevant constraints, but we also calculate the percentage of nodes that conform to their constraints.

3.4. Front-end

We implemented a front-end where the user can choose a *knowledge graph* as well as a type of knowledge graph. In addition, the user can also set a limit. As output, *ShEx* constraints as well as a validation of those constraints are given. The constraints can be edited by the user and those edited constraints can be revalidated. If a node is deemed invalid, a reason is given, e.g. "Cardinality violation (min=1): 0" The user can download the subset of the graph which was validated. The interaction between user, front-end and server can also be seen in Fig. 2.

check whether
this is what
we are doing
in the finished
version



explain this
in more detail,
maybe also put
the explanation
in the query

explain how
different limits
influence the
output

update and
scale sequence
diagram and
refer to it

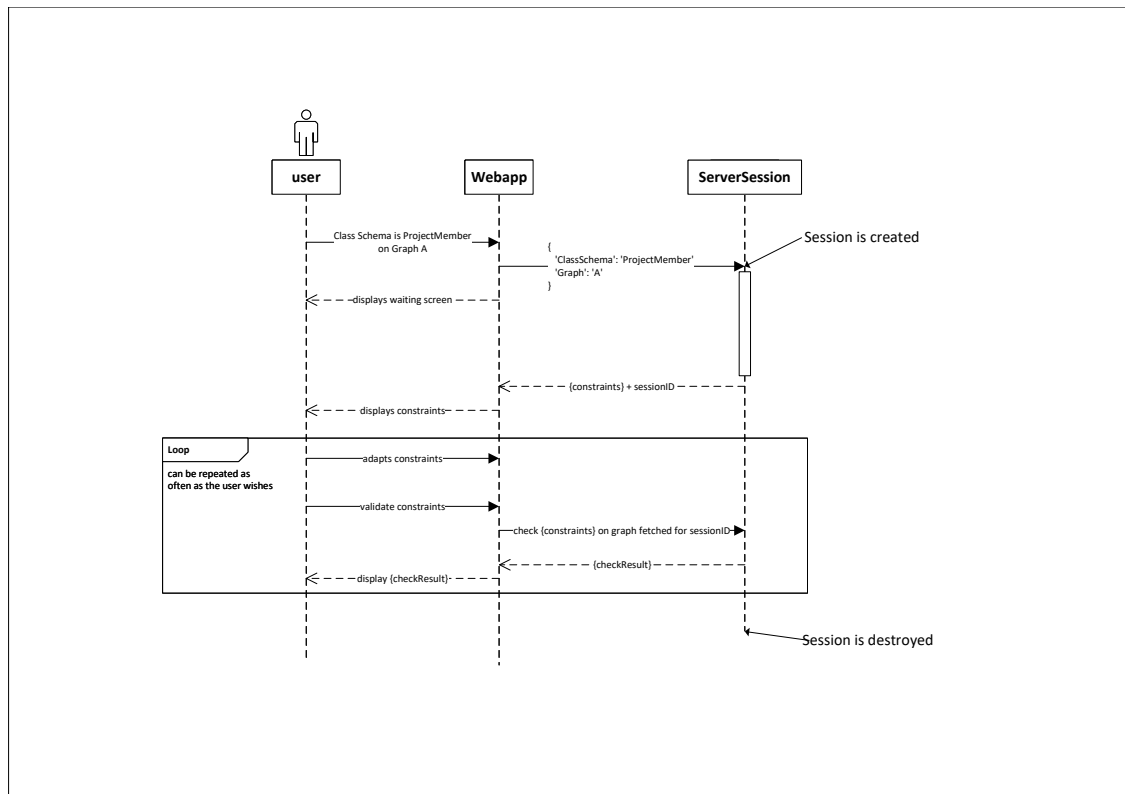


Figure 2: Sequence diagram showing the interaction between web application, user and server

4. 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. User can also edit constraints.

Missing figure

Maybe add small figure that shows workflow of project here? Something similar like we did in presentation but more professional?

explain this limit in more depth, maybe in front-end?

4.1. 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

describe results of benchmark tests here

Possible future work could be: more datasets, more possibilities for user inputs

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.

Should we add proper SPARQL endpoints here? Might not be possible?

5. Evaluation

add benchmarks here

5.1. Methodology

ji labeled all the included graphics with h!, when we have finished the report we might want to make it so, that one image is on the top and one on the bottom, if 2 pages are on the same page, for example; For taking the measurements the application was started locally on our hardware, to minimize side-effects of other applications running on the virtual machine where the live-instance is deployed. The JVM was additionally setup 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.

check what Ewin said concerning Evaluation on meeting 20.01.2022

5.2. Runtime

Figures 3 and 4 show our measurements we obtained by tuning the *LIMIT* input parameter, therefore tuning the size of the start-node subset, from which connected nodes are fetched. All the measurements are shown in tables 1 and 2.

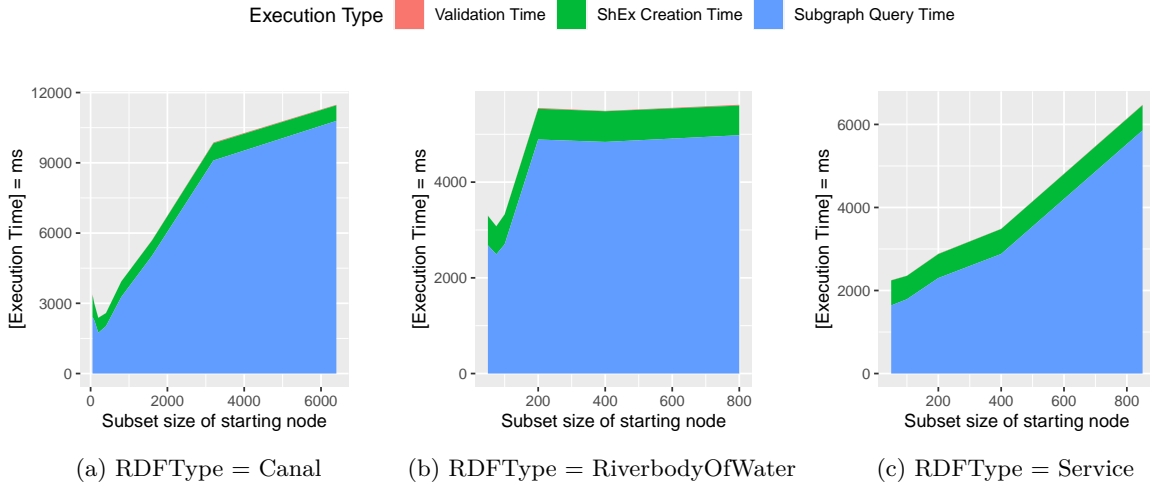


Figure 3: Execution times per RDFType, per size of start-node subset on RiverbodyOfWater dataset

The results shown in 3 were to be expected. First of all, the runtime of fetching the wanted subset of the graph is considerably larger than the time needed to create the *ShEx* constraints, or to validate the constraints on the graph. This can also be seen in figure 5, where we didn't provide any limit. Secondly, the smaller the *LIMIT* the smaller the runtime. This becomes especially clear in figures 3a and 3c.

To understand the behaviour shown in figure 3b, we want to look at figure 4, which shows the same runtimes, but grouped by the number of triples, contained in the fetched graph, on which the constraints

are created. Unlike in figures 4a and 4c, the maximum number of triples (shown in the x-coordinate in figure 4b), is 1769. This is also the amount of triples contained in the graph, without providing any limit, showing us that providing a larger limit than 200, won't enrich the fetched graph, therefore keeping the time almost constant, in regards to the *LIMIT* parameter.

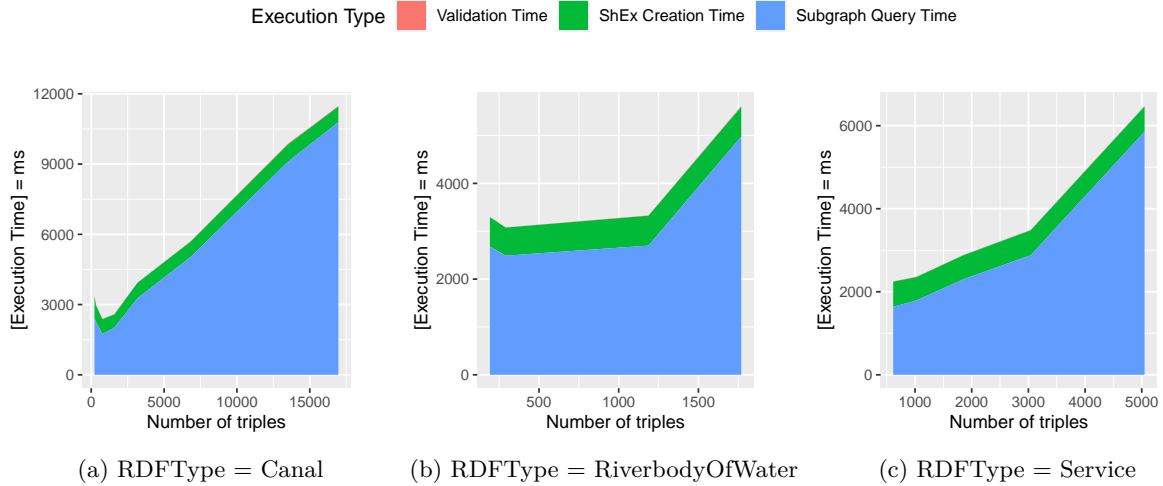


Figure 4: Execution times per RDFType, per number of triples on RiverbodyOfWater dataset

Figure 5 shows the runtime without any provided limit to the query fetching the subgraph. Note the much larger time, needed to make the query, despite having the same amount of triples when providing a large enough *LIMIT*.

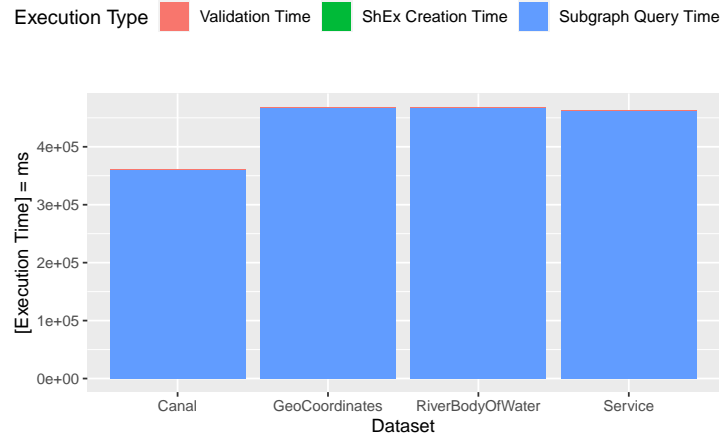


Figure 5: Execution times per RDFType, queried on full graph of the RiverbodyOfWater dataset (containing 49915 triples)


```

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 6: Anonymous Nodes in Turtle File

5.3. Correctness

5.4. ShEx Validation

6. Conclusion

References

- [1] Vue.js, documentation. <https://v3.vuejs.org/>.
- [2] J. van Dam. Rdf2graph. <https://github.com/jessevdam/RDF2Graph/>, 2022. Accessed: 2022-01-16.

A. Contribution Statements

Please write down a short contribution statement for each member of your group. You may evaluate the contribution along the three common categories: i) conception (i.e., problem framing, ideation, validation, and method selection), ii) operational work (e.g., setting up your tech stack, algorithm implementation, data analysis, and interpretation), and iii) writing & reporting (i.e., report drafting, literature review, revision of comments, presentation preparations, etc.).

B. Appendix

You may use appendices to include any auxiliary results you would like to share, however cannot insert in the main text due to the page limit.

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

Which challenges did we face during the implementation? (Maybe dept of SPARQL query, outdated RDF2Graph)

Did we achieve what we wanted to do? How well and reliably does the framework work?

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	850	5050	5856	606	12

Table 2: Execution times per RDF-Type, limited size of start-node subset (using the RiverBodyOfWater dataset)