Knowledge Graphs

Knowledge Graph of Scientific Research Papers

Master in Data Science, FIB, UPC
Semantic Data Management

May 18th, 2023

Marçal García Boris - Odysseas Kyparissis

Table of Contents

A. Introduction

For the sake of this laboratory project, as the knowledge graph needed to be constructed will include data about scientific research papers, the following website was used for mining the data: https://www.aminer.org/citation. From this website the 14th version of the dataset was used (DBLP-Citation-network V14). As the size of the file was huge, and several preprocessing techniques needed to be applied, two preprocessing scripts were created, which can be found inside the zipped file delivered, under the folder *preprocessing_scripts* (and here as well). More information about the preparation of the data can be found in section: B2. ABOX Definition. The final software produced for this laboratory can be found inside the zipped file delivered with the name *12E-B2-GarciaKyparissis-Java-Code.zip* and in GitHub[1] by following this link. Once the Jena package is missing from the zipped file for size purposes, it is preferred to clone the code from the repo.

B. Ontology Creation

B1. TBOX Definition

To begin with, for creating the TBOX of the solution, we made use of Protégé[2]. Protege, is an open-source ontology editor and framework for building intelligent systems. For the creation of the TBOX, we used OWL as the knowledge graph language. Figure 1. depicts the visual representation of the proposed TBOX design.

As the representation of Figure 1. is not of high quality, the visual representation of the graph can be found here and inside the folder *visualization* of the delivered zip file (download the file and open it via a WebBrowser).

During the design of the TBOX, several decisions and assumptions needed to be taken in order to create a representation that follows description logic. More specifically, in order to get a clearer view and understanding of the class hierarchy that was designed, one can take a look at Figure 2.

As it is depicted, the main classes include *Area, ConferenceProceedings, JournalVolume, Paper, Person, Review,* and *Venue*. Class *Paper* consists of two subclasses *AcceptedPaper* (referred to as a Publication in the statement) and *RejectedPaper*. Each of these subclasses further includes four more subclasses namely *DemoPaper*, *FullPaper*, *Poster* and *ShortPaper*, respectively. Additionally, *Author, Chair, Editor* and *Reviewer* are subclasses of *Person.* Moreover, class *Review* includes *NegativeReview* and *PositiveReview* subclasses. It's important to mention here that we consider that two reviewers share the same final review. Thus, a paper

[1] https://github.com/
[2] https://protege.stanford.edu

will have two reviewers but only one review. As for the *Venue* class, it was divided into *Conference* and *Journal* subclasses, where *Conference* includes *ExpertGroup, ReguralConference, Symposium* and *Workshop*. The specific design allows the inclusion of all the restrictions requested by the design description.

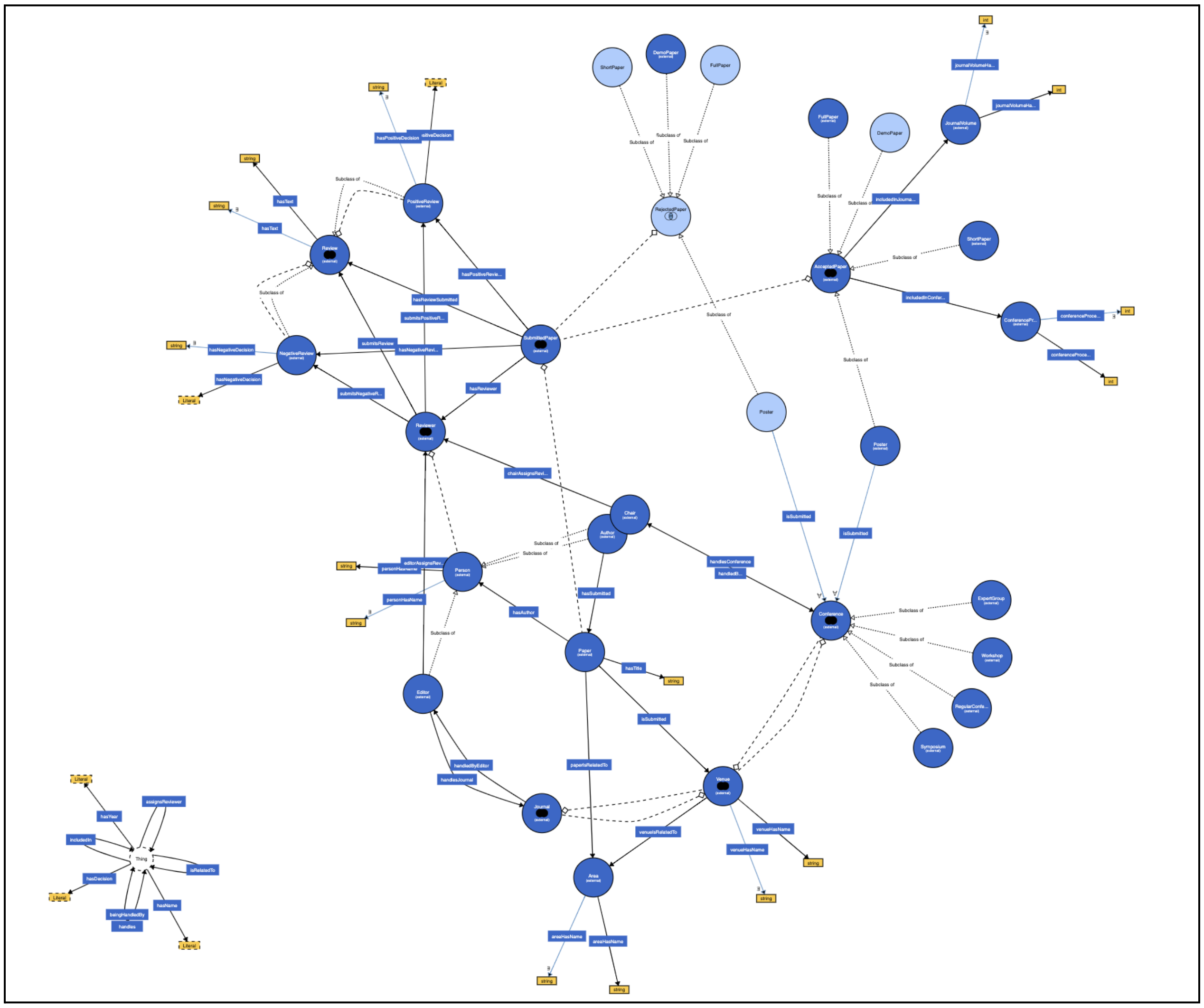


Figure 1. Visual Representation of the Knowledge Graph

The creation of a superclass named *Person,* which includes as subclasses *Author*, *Chair*, *Editor*, and *Reviewer* allows the individuals of the subclasses to have common attributes, e.g.: name, surname, date of birth, nationality, etc. In that way, the representation of people in the knowledge graph is more structured, generic, and allows easier graph expandability and knowledge extraction, for the users of the database. The same is true for all the super classes generated in this solution.

Furthermore, even though it is generally expected that an editor of a journal should have a strong background in the field covered by the journal, which often includes having published research in that field, as it is not an absolute requirement of the statement, we don’t consider editors as subclassOf Authors. We follow the same idea about Reviewers as well as Chairs. The main reason for applying these assumptions, is that once the data preprocessing part was challenging, we considered spending more time in optimizing the design of the ontology with the usage of description logics, rather than cleaning the data.

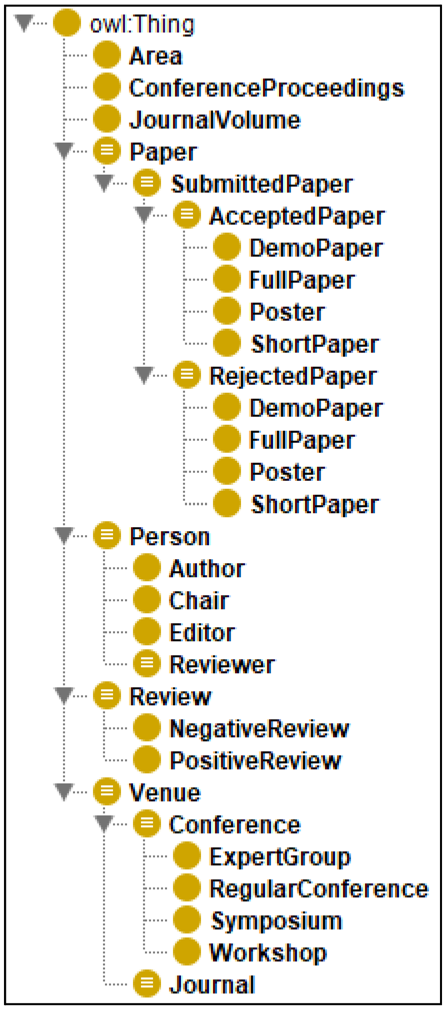


Figure 2. Hierarchy of the Classes in TBOX

When designing the Object Properties, in order to better set the constraints, in most cases we created subproperties inside a super one. This can be seen, for example, in *includedIn*, which has sub-properties *includedInConferenceProceedings* and *includedInJournalVolume*, which both have as domain *AcceptedPaper* but the range can vary between the classes *ConferenceProceeding* and *JournalVolume*. The same idea was followed for the creation of the Data Properties.

Moreover, as the dataset used consisted of already published research papers we considered that all of them are of class *SubmittedPaper*, but we included some fake data in order to include in the final knowledge graph both accepted and rejected papers. Ultimately, as it was not a critical step to generate fake review text for the text attribute of the class *Review,* the text of the abstract of each paper was used. Last step of the TBOX definition was the extraction of the

TBOX file in RDF/XML format, which supports the full restrictions and constraints that can be applied with OWL. The file can be found inside the folder *TBOX_ABOX_files* of the zip file delivered, as well as by clicking on the link just provided.

B2. ABOX Definition

Once the TBOX was created with Protégé and exported into RDF/XML format, the ABOX definition took place. As mentioned before in section: A. Introduction, the dataset used was preprocesses, following two preprocessing steps. We introduce one different file for each step, and they can be found inside the zipped file delivered, under the folder *preprocessing_scripts* (and here as well). During the first preprocessing step, the json files downloaded from aminer.org[3], were transformed into a single CSV file. During the second preprocessing step a final CSV file was prepared in order to include extra information requested for the specific solution. Several information that was finally used was artificially generated, in order to fulfill the design requested as well as the final queries. One thing to mention here is that, for the generation of *Editor* and *Chair* names (as well as their IDs) random names were used, and not ones from the *Author* column.

In order to create all the restrictions and to manipulate the triples required for the solution of this laboratory project, the usage of Jena API[4] with Java[5] was necessary. It is worth mentioning here that the way we prepared the data for loading them inside the ABOX, is not close to an optimal approach. Due to the reason that the final CSV file includes for each paper as many rows as the authors, by replicating the values of the remaining columns to all the rows, lead to the creation of several extra if-else statements inside the ABOX code. Preparing the data into separate files for each concept (Class, Property) would lead to a more robust solution, but when we understood that it was already too late.

Thus, for the creation of the ABOX, the RDF API as well as the Ontology API were used. During the development phase of the ABOX, the *Owl Reasoner* was used, in order to validate the insertion of triples in the ABOX model. Both the reasoner used, and the ABOX and TBOX models used in Java, were configured to follow *OWL_MEM_MICRO_RULE_INF* inference. The *OWL_MEM_MICRO_RULE_INF* reasoning profile in Jena includes a subset of inference rules from the *OWL 2 RL* (RDF Rules Language) rule set. This subset focuses on providing basic reasoning capabilities while maintaining computational tractability. The reasoning tasks are as follows:

- **Class Hierarchy**: The reasoner infers subclass and superclass relationships based on the available class assertions and class expressions.
- **Property Hierarchy**: The reasoner infers subproperty and superproperty relationships based on the property assertions and property expressions.
- **Property Domain and Range**: The reasoner infers the domain and range of properties based on the available property assertions.

[3] https://www.aminer.org/
[4] https://jena.apache.org/
[5] https://www.java.com/en/

- **Property Characteristics**: The reasoner infers property characteristics such as symmetry, reflexivity, and transitivity based on the available property assertions.
- **Property Inversion**: The reasoner infers inverse property relationships based on the available assertions.
- **Property Chaining**: The reasoner infers property chaining relationships based on the available property assertions.
- **Class Assertion**: The reasoner infers class membership assertions based on the available subclass relationships and property assertions.

In that way, during the development phase of the solution, we could be sure that the data inserted does not produce any error and their integrity is correct. The general methodology followed during the development of the software for the creation of the ABOX is the following:

- Create the OWL Reasoner
- Parse the TBOX file to create the TBOX model in Jena, in order to have access to the Classes and Properties directly with API calls
- Create an empty ABOX model
- Parse the CSV file containing the instance data
- Clean the strings in order to be ready to be loaded into the ABOX (e.g. remove spaces, or replacing special characters with **"_"**)
- Create the Individuals for each different class in ABOX, in the lowest hierarchical level possible, once the highers are generated automatically by inference
- Create the properties for the individuals (both Object and Data properties) in ABOX again in the lowest hierarchical level, for the same reason as before.
- Once all the nodes were generated and all the instances were included as triples to the ABOX, export it as a file with ***.nt*** extension.

The final ABOX ontology file can be found inside the folder *TBOX_ABOX_files* of the zip file delivered (and here as well). The code of Java that generates the ABOX, can be found in GitHub by following this link and inside the zip file with the name *12E-B2-GarciaKyparissis-Java-Code.zip*. Once the Jena package is missing from the zipped file for size purposes, it is preferred to clone the code from the repo.

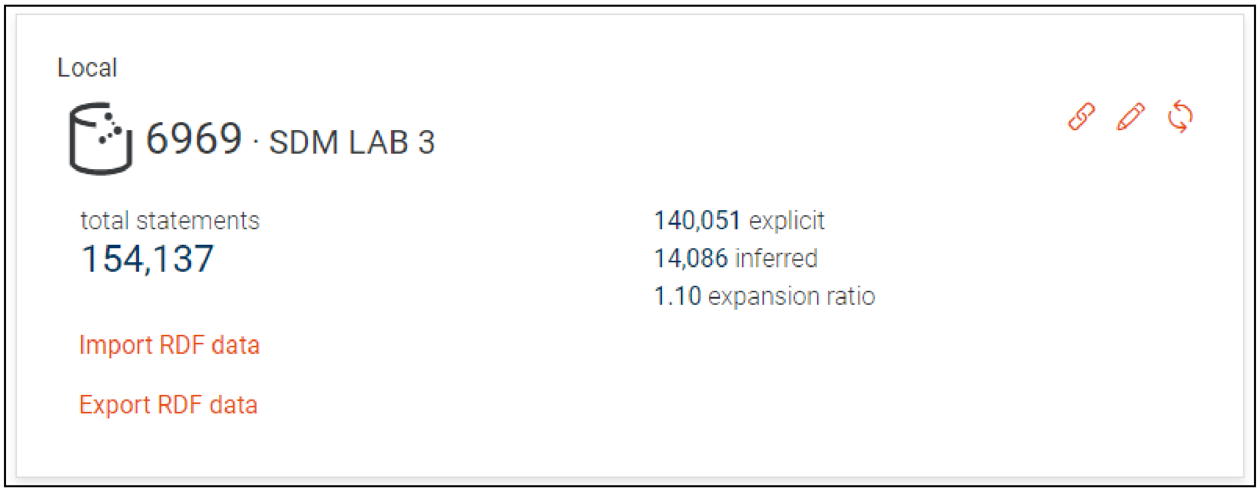
B3. Create the Final Ontology

As explained in section: B2. ABOX Definition, the links were created programmatically with the usage of Jena API, and they can be found in the code final file of ABOX and the code provided. As for the inference regime entailment that we considered, all the information is also provided in the previous section. For example, some of the *rdf:type* links that were saved to explicitly generate new knowledge thanks to reasoning, is that the domain of the property areaHasName is of Class *Area*. The same is true for the domain and range of the object property *isSubmittedTo,* showing that the domain is of Class *Paper* and the range of Class *Venue*. But for the properties that include subproperties, we were always using the lower level of hierarchy while creating the triples, in order to generate all the available knowledge with the minimum statements and insertions possible.

Consequently, when both the ABOX and TBOX files were generated, the importing of them to the GraphDB software took place. By following the instructions of *Section A* of the statement the creation of the knowledge graph was completed with ease.

It is important to note that we used the option *RDFS-Plus (Optimized)* during the importing of the knowledge graph, in order to enable reasoning for GraphDB. Besides this configuration the default values were used for the remaining options. For the base IRI we used the following one: <http://www.semanticweb.org/sdm>, which is the same that was used by Protege while generating the TBOX. For the *Target graphs* option we chose the Named *graph* and entered the following in the text box: http://localhost:7200/sdm/.

After importing the files and creating the graph, we were able to explore it and calculate the requested statistics. First of all, as depicted in Figure 3., the database includes 154,137 statements in total, 140,051 of which are explicitly generated and 14,086 are inferred. The ratio is small due to the nature of the data that were used for the generation of the triples and the ABOX. Once the *Chair* and *Editor* Instances of the dataset were not considered *Authors* and they were artificially generated (independently of the available authors) as well as the fact that we only included 100 papers, lead to this result.


Local

6969 · SDM LAB 3

total statements
154,137

140,051 explicit
14,086 inferred
1.10 expansion ratio

Import RDF data

Export RDF data

Figure 3. Repository Information

Statistics and Summary of the Knowledge Graph

In order to compute a summary and basic statistics for the instances of the graph, we used the following queries which aim in computing the total number of classes, properties, and specific instances both from classes and properties respectively:

This first query computes the total number of Classes:

```
Unset
PREFIX owl: <http://www.w3.org/2002/07/owl#>
```

```
SELECT (COUNT(DISTINCT ?class) AS ?numClasses) WHERE {
  ?class a owl:Class.
}
```

Here the total number of distinct properties is calculated:

```
Unset
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT (COUNT(DISTINCT ?property) AS ?numProperties) WHERE {
  ?property a rdf:Property.
}
```

The following query retrieves the total amount of distinct instances for each class that is a subclass of the specified *class_URI*.

```
Unset
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?className (COUNT(DISTINCT ?instance) AS ?numInstances)
WHERE {
  ?instance rdf:type ?class .
  ?class rdfs:subClassOf* <class_URI>.
  FILTER (!isBlank(?instance))
}
GROUP BY ?className
```

Finally, this last query retrieves the total number of distinct triples for each property *property_URI*.

```
Unset
SELECT DISTINCT ?propertyName (COUNT(*) AS ?numTriples) WHERE {
  ?s <property_URI> ?o .
  FILTER (!isBlank(?s) && !isBlank(?o))
}
GROUP BY ?propertyName
```

Finally, we can sum up some of the experiments done into the following table:

MEASURE	VALUE
Total Number of Classes	183
Total Number of Properties	74
Total Instances of Class “Paper”	100
Total Instances of Class “Person”	618
Total Instances of Class “Review”	100
Total Instances of Class “Venue”	97
Total Instances of Class “Author”	318
Total Papers with “PositiveReview”	51
Total Papers with “NegativeReview”	49
Total Instances of Property “hasAuthor”	100
Total Instances of Property “hasSubmitted”	318
Total Instances of Property “includedInConferenceProceedings”	28
Total Instances of Property “includedInJournalVolume”	23

Ultimately, by exploring GraphDB and its capabilities we generated Figure 4., which presents the relationships between the classes of the knowledge graph.

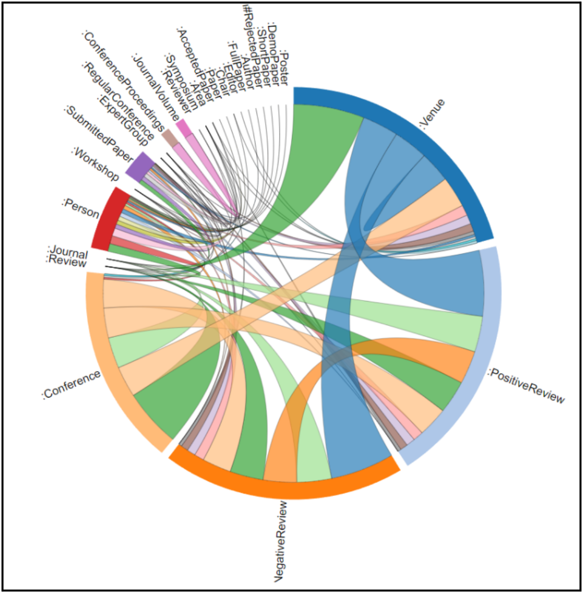


Figure 4. Repository Information

B4. Querying the Ontology

In this section of the report the requested queries are included. The results of the queries can be found inside the folder *query_results* of the delivered zipped file, named *q1, q2, q3, q4* plus the naming convention requested, respectively (and here).

B4.1. Find all Authors

In this first query we aim to find all the instances of the variable *?author* that belongs to the class *rdf:Author*.

```
Unset
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sdm: <http://www.semanticweb.org/sdm/>
SELECT DISTINCT ?author WHERE {
  ?author rdf:type sdm:Author
}
```

B4.2. Find all properties whose domain is Author

The second query selects all properties *?property* that has the rdfs:domain of http://semanticweb.org/sdm/Author. Thus, it retrieves properties whose domain is specifically the *Author* class. In this case we'll only obtain one property which is *hasSubmitted*.

```
Unset
PREFIX sdm: <http://www.semanticweb.org/sdm/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?property WHERE {
  ?property rdfs:domain sdm:Author.
}
```

B4.3. Find all properties whose domain is either Conference or Journal

The third query searches for triples where the *rdfs:domain* property of a property *?property* is either *sdm:Conference* or *sdm:Journal*. In order to get the combined result from both parts into a single result set we use the UNION operator.

```
Unset
PREFIX sdm: <http://www.semanticweb.org/sdm/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
SELECT ?property WHERE {
  {
    ?property rdfs:domain sdm:Conference.
  }
  UNION
  {
    ?property rdfs:domain sdm:Journal.
  }
}
```

B4.4. Find all the papers written by a given author that where published in database conferences

In this last query, we first had to check which Authors wrote some papers that were Accepted, thus, are publications. From all the authors retrieved, we choose the one with the name *Fu_Jin* and we check in which conferences they were published. We also retrieve the area of these conferences.

```
Unset
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sdm: <http://www.semanticweb.org/sdm/>
SELECT ?paper ?author ?conference ?area
WHERE {
  ?paper rdf:type sdm:AcceptedPaper ;
         sdm:hasAuthor ?author .

  ?author rdf:type sdm:Author ;
          sdm:hasName "Fu_Jin" .

  ?paper sdm:isSubmitted ?conference.

  ?conference sdm:isRelatedTo ?area.
}
```