University of Stuttgart
Germany

Text Technology

Individual Report

From Ontology To GraphDB

Mahdi Jafarkhani

Matriculation Number: 3679412

August 16, 2024

1 Project Overview

In this project, we aim to read an ontology given in RDF-XML format based on DBPedia [1], into a Graph database model to perform graph queries, like neo4j. In the vocabulary of Metadata Engineering, an ontology, as a data model is given, and we want to fetch instances of this model using DBPedia endpoint. Table 1 illustrates the mapping between two data models. Table 2 illustrates the whole workflow process of this project.

Ontology Model	**Graph Model**
Classes	Nodes
Data Type Properties	Node Attributes
Object Properties	Edges

Table 1: Conversion between Ontology and Graph Database Model

Step	**Process**	**Input**	**Output**	**Category**
1	Metadata Extractor	RDF/XML Ontology	Binary Metadata	Collect, Prepare
2	DBPedia Accessor	Binary Metadata	JSON Data	Collect, Prepare
3	Cypher Generator	JSON Data	Cypher File	Access

Table 2: Workflow

1.1 Ontology Metadata Extractor

In this project, our goal is to read an ontology in RDF-XML format, specifically based on DBPedia, and load it into a graph database model, such as neo4j, to enable graph queries. These metadata include **Classes**, **Data Type Properties**, and **Object Properties**, which are extracted using XPATH. The output of this step is some Python object files, which are stored as binary.

1.2 DBPedia Endpoint Accessor

In the second step, now we have all the metadata that are required to fetch data from DBPedia. We use SPARQL endpoint of DBPedia and create automatic scripts to get data from the online database. These data are stored in JSON format. Since DBPedia endpoints restrict query outputs to only 10,000 records per request, we use pagination to retrieve all data. The queries are optimized to retrieve only related data. For example, if we have two classes of `Person` and `Film`, and one relation between them such as `director`, we only retrieve those persons who are involved in at least one film. In other words, the output does not have any dangling nodes (nodes without any relations incoming into or outgoing from them).

1.3 Cypher Files Generator

In the last step, we read the JSON data from the previous step and convert them into Cypher script files. The final output is only one file that is ready to be run on a neo4j database

[1] DBpedia is a community-driven project that aims to extract structured information from Wikipedia

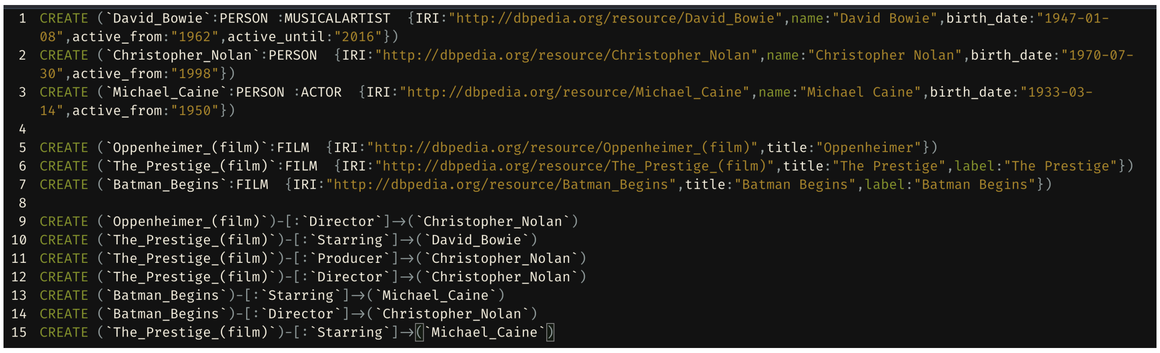
2 Challenges

2.1 DBPedia SPARQL Endpoint Restriction

One challenge we encountered during the **collection** phase of our project is that the DBpedia SPARQL endpoint imposes a limit of 10,000 records per request. This restriction is a common security measure since the endpoint is openly accessible and free to use on the web. To overcome this limitation, we implemented a pagination mechanism in SPARQL, which helps break down query results into smaller, more manageable chunks—particularly useful for large datasets. The pagination process in SPARQL primarily involves two key keywords: `LIMIT` and `OFFSET`. For instance, in our ontology `Movie.owl` example, the `Person` class contains 43,625 instances. To retrieve all records, we first determine the total number of instances with a query. We then divide this number by the limit (10,000) and execute five additional queries using `OFFSET` values of 0, 10,000, 20,000, 30,000, and 40,000. This approach allows us to retrieve all 43,625 records with six requests to the endpoint.

2.2 Class Hierarchy in Neo4J

Another challenge that is worth noting is supporting class hierarchy in ontology. This challenge could be classified in the **prepare** aspect of our project. Back to our `Movie.owl` example, as depicted in Figure 2, we support ontologies that have class hierarchy at the first level. To reflect this concept in Neo4j we use multiple-node-labling in the final cypher file. For example, `David_Bowie` is a `Person` and also a `MusicalArtist`. To reflect this fact in our database, the node corresponding to it has two labels, as depicted in the first line of Figure 1. This is also the case for node `Michael_Caine` which is also a `Person` but only `Actor`. Other nodes, like `Christopher_Nolan` have only one label, which is `Person`.



```
CREATE (`David_Bowie`:PERSON :MUSICALARTIST  {IRI:"http://dbpedia.org/resource/David_Bowie",name:"David Bowie",birth_date:"1947-01-08",active_from:"1962",active_until:"2016"})
CREATE (`Christopher_Nolan`:PERSON  {IRI:"http://dbpedia.org/resource/Christopher_Nolan",name:"Christopher Nolan",birth_date:"1970-07-30",active_from:"1998"})
CREATE (`Michael_Caine`:PERSON :ACTOR  {IRI:"http://dbpedia.org/resource/Michael_Caine",name:"Michael Caine",birth_date:"1933-03-14",active_from:"1950"})

CREATE (`Oppenheimer_(film)`:FILM  {IRI:"http://dbpedia.org/resource/Oppenheimer_(film)",title:"Oppenheimer"})
CREATE (`The_Prestige_(film)`:FILM  {IRI:"http://dbpedia.org/resource/The_Prestige_(film)",title:"The Prestige",label:"The Prestige"})
CREATE (`Batman_Begins`:FILM  {IRI:"http://dbpedia.org/resource/Batman_Begins",title:"Batman Begins",label:"Batman Begins"})

CREATE (`Oppenheimer_(film)`)-[:`Director`]->(`Christopher_Nolan`)
CREATE (`The_Prestige_(film)`)-[:`Starring`]->(`David_Bowie`)
CREATE (`The_Prestige_(film)`)-[:`Producer`]->(`Christopher_Nolan`)
CREATE (`The_Prestige_(film)`)-[:`Director`]->(`Christopher_Nolan`)
CREATE (`Batman_Begins`)-[:`Starring`]->(`Michael_Caine`)
CREATE (`Batman_Begins`)-[:`Director`]->(`Christopher_Nolan`)
CREATE (`The_Prestige_(film)`)-[:`Starring`]->(`Michael_Caine`)
```

Figure 1: A part of the Cypher file that is created at the end of the workflow

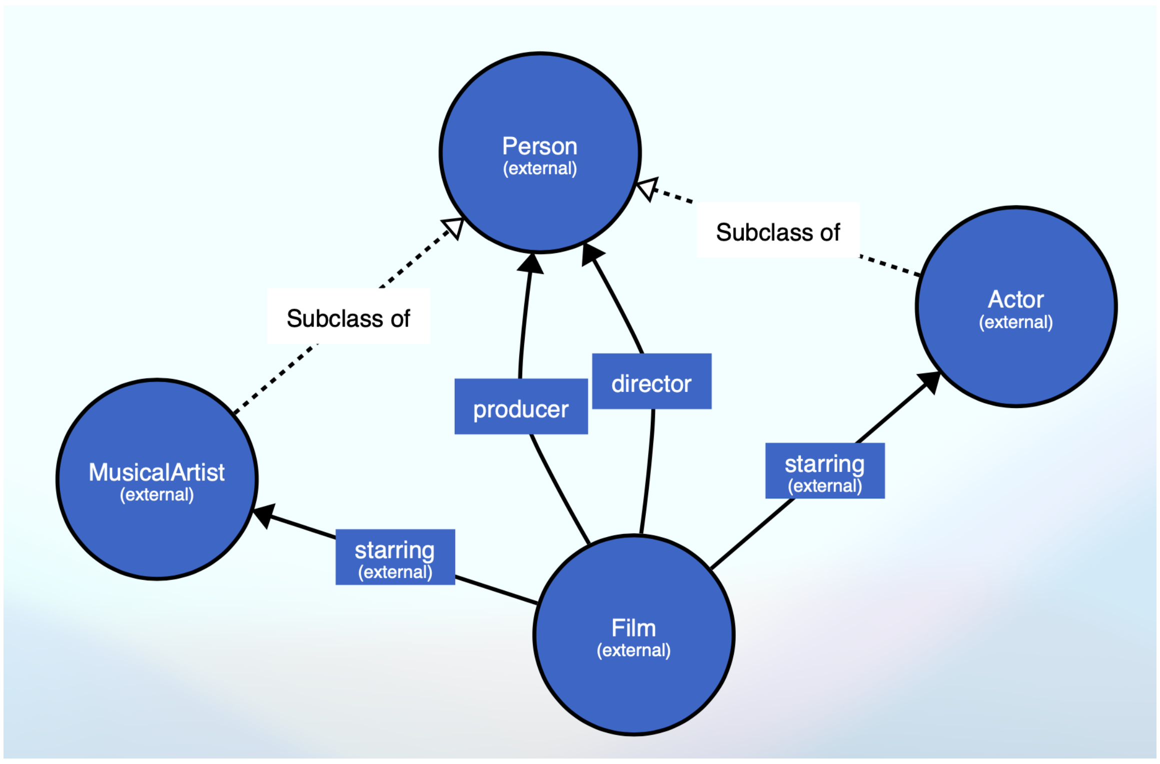


Figure 2: A visualization of the Movie ontology