Lab Session #04

Introduction

Welcome to Lab #04. This week's lab is about learning the *SPARQL query language* for knowledge graphs. As always, if you did not finish any of the tasks from the previous week(s), make sure you catch up with any remaining tasks, since some of the new tasks build on previous work.

Before diving into SPARQL, let's briefly touch upon the concept of *graph queries*. In the realm of data science and knowledge representation, graph queries allow us to retrieve and manipulate data structured in the form of graphs - think of nodes representing entities and edges depicting relationships between them. This approach is paramount in navigating complex networks of information, from social media interactions to biological ecosystems. SPARQL, or *SPARQL Protocol and RDF Query Language*, is designed specifically for querying and managing data stored in the Resource Description Framework (RDF) format. Mastery of SPARQL enables us to efficiently extract meaningful insights from vast, interconnected datasets, making it an indispensable tool in the development of intelligent systems.

Follow-up Lab #03

Here's an example solution for the FOCU university schema & data from the previous lab. Of course, your solution might look slightly different. If you're confused about any part of it, just ask in the Moodle Discussion forum!

Task #1: SPARQL 101

Here is a nice SPARQL cheat sheet (well, more like cheat slides) that are helpful for getting a quick overview (note that we did not discuss all the details of SPARQL in class). To learn SPARQL it's best to experiment with a number of different queries. Many (but not all) public knowledge graphs provide a public SPARQL interface (a so-called *SPARQL endpoint*), for example this one at DBpedia.

Note that these public, open endpoints are typically very restrictive in terms of query result size, execution time, and query memory use, so don't be surprised if you get an error message instead of a result. Sometimes it helps to retry a query or limiting the result size (e.g., with a `LIMIT 50`).

Task #1.1: DBpedia

To gain an impression on how powerful graph queries can be, here are two examples that you can try out using DBpedia's public SPARQL endpoint:

1. Musicians who were born in Berlin:

```
2.  SELECT ?name ?birth ?description ?person
3.  WHERE {
4.        ?person a dbo:MusicalArtist .
5.        ?person dbo:birthPlace dbr:Berlin .
6.        ?person dbo:birthDate ?birth .
7.        ?person foaf:name ?name .
8.        ?person rdfs:comment ?description .
9.        FILTER (LANG(?description) = 'en') .
10. } ORDER BY ?name
```

This query retrieves a list of musical artists born in Berlin, including their names, birth dates, and descriptions in English, ordered by name. It demonstrates how SPARQL can filter and sort data from multiple properties within a graph.

11. Soccer players, who are born in a country with more than 10 million inhabitants, who played as goalkeeper for a club that has a stadium with more than 30.000 seats and where the club's country is different from the player's birth country:

```
12. SELECT DISTINCT ?soccerplayer ?countryOfBirth ?team ?countryOfTeam
    ?stadiumcapacity
13. WHERE {
14.      ?soccerplayer a dbo:SoccerPlayer ;
15.            dbo:position|dbp:position
    <http://dbpedia.org/resource/Goalkeeper_(association_football)>;
16.            dbo:birthPlace/dbo:country* ?countryOfBirth ;
17.            dbo:team ?team .
18.      ?team dbo:capacity ?stadiumcapacity ;
               dbo:ground ?countryOfTeam .
19.      ?countryOfBirth a dbo:Country ;
               dbo:populationTotal ?population .
20.      ?countryOfTeam a dbo:Country .
21.      FILTER (?countryOfTeam != ?countryOfBirth)
```

```
    FILTER (?stadiumcapacity > 30000)
    FILTER (?population > 10000000)
} ORDER BY ?soccerplayer
```

This sophisticated query illustrates SPARQL's ability to navigate through interconnected properties and apply multiple filters, identifying goalkeepers who meet specific criteria regarding their birthplace, team's stadium capacity, and more, showcasing the intricate queries possible with SPARQL.

You can see how intelligent assistants like Watson, Siri, Alexa etc. are able to answer so many questions, by querying their knowledge graphs.

Now, try to write your own queries to determine:

1. All *universities located in Canada*, with their city and *optionally* (if it exists) their home page.
2. All people who *studied at Concordia University* (and are listed in DBpedia), together with their description (in English or any other language you prefer). Hint: look for the *Alma Mater* predicate and make sure you understand its domain & range.

Additional Hints:

- For the first task, start by identifying the RDF type that corresponds to universities and then narrow your search to those located in Canada. Use the `dbo:country` and `dbo:city` properties to help specify the location.
- For the second task, exploring DBpedia to understand how Concordia University is represented (its resource URI) will be key. Use this URI to filter individuals who have a relationship indicating they studied there.

These exercises are designed to enhance your ability to formulate precise queries and extract specific information from a vast dataset. Good luck!

Task #2: Your own SPARQL Server

As discussed in the lecture, setting up your own SPARQL *endpoint* is a pivotal step in working with knowledge graphs for Intelligent Systems (IS) projects. At the core of such projects are **triplestores** and **query engines**. *Triplestores* are specialized database management systems designed for storing and retrieving triples of information, where each triple represents a fact in the form of a *subject-predicate-object* relation. This structure is essential for efficiently managing the complex, interconnected data found in knowledge graphs. *Query engines*, on the other hand, interpret and execute the SPARQL queries you write, fetching data from the triplestore based on your query's criteria.

To facilitate this, we'll be using Apache Fuseki, a highly regarded open-source SPARQL server that's part of the Apache Jena project. Apache Jena is a comprehensive framework for building semantic web and linked data applications. It includes a powerful programming API and a SPARQL query engine. Fuseki, as its server component, makes it easy to deploy your triplestore and expose it via a SPARQL endpoint, allowing for the querying and updating of your RDF data over HTTP. Opting for an open-source engine like Fuseki for our initial experiments and the course project offers several advantages. It provides a cost-effective solution without the complexities and potential limitations of commercial or cloud-based services. Moreover, it encourages a deeper understanding of the underlying technology and fosters a hands-on approach to learning how SPARQL servers operate.

However, the world of SPARQL servers extends beyond open-source solutions. For commercial or larger-scale projects, several cloud-based and commercial engines offer advanced features, scalability, and support. Notable examples include Amazon Neptune, a fully managed graph database service that supports SPARQL for querying, and Stardog, which provides a powerful semantic graph database with extensive querying capabilities. GraphDB is another sophisticated option, offering robust support for semantic graph databases and a user-friendly interface. Each of these platforms has its unique strengths, catering to different requirements and scales of operation.

Exploring these alternatives can provide valuable insights into the range of possibilities and considerations when selecting a SPARQL server for specific project needs. For our initial experiments and the course project, starting with Apache Fuseki allows us to focus on the fundamentals. Still, being aware of these commercial and cloud-based solutions will prepare you for future projects that might demand their advanced capabilities.

Task #2.1: Getting started with Fuseki

1. Start by downloading the `fuseki` binary distribution from https://jena.apache.org/download/index.cgi. This package contains all you need to set up and run your Apache Fuseki server.
2. Unpack the archive and make the server script executable. These steps prepare the server's environment on your machine, allowing you to run Fuseki locally:

3. `> tar xf apache-jena-fuseki-4.10.0.tar.gz`
4. `> cd apache-jena-fuseki-4.10.0`
5. `> chmod u+x fuseki-server`

Please note, the commands provided above are tailored for Linux. If you're using Windows or MacOS, the installation steps will differ slightly. It's important to consult the Apache Fuseki documentation or relevant system guides to adjust commands accordingly for your operating system. On MacOS, you can use the

Terminal, while on Windows, you'll use a GUI or PowerShell to extract the files and may not need to change the script's execution permissions.

6. Now, launch the server. Ensure you have a Java Development Kit (JDK 11) installed, as Fuseki runs on the Java platform. This command starts the Fuseki server, which will listen for requests on port 3030:

```
7.  > ./fuseki-server
8.  [2020-02-09 09:55:15] Server     INFO  Apache Jena Fuseki 3.17.0
9.  [2020-02-09 09:55:15] Config     INFO  FUSEKI_HOME=/home/rene/fuseki/apache-jena-
    fuseki-3.17.0/.
10. [2020-02-09 09:55:15] Config     INFO  FUSEKI_BASE=/home/rene/fuseki/apache-jena-
    fuseki-3.17.0/run
11. [2020-02-09 09:55:15] Config     INFO  Shiro file: file:///home/rene/fuseki/apache-jena-fuseki-
    3.17.0/run/shiro.ini
12. [2020-02-09 09:55:17] Server     INFO  Started 2020/02/09 09:55:17 EST on port 3030
```

Access Fuseki's web interface by navigating to http://localhost:3030/ in your browser. This interface is your control panel for managing datasets and queries.

13. Before you can do anything else, you must create a *Dataset*, so select the first open above:

Give your dataset a name and use the "In-memory" option (this means your triples will not be stored persistently, but this is fine for some first experiments).

14. You should now see your new dataset under "existing datasets" and can start to upload triples to populate it with data. Use your university (FOAF/FOCU) triples from two weeks ago (or any other triple file you have):

15. Now you are ready to send SPARQL queries to your server: go to the *query* tab and start querying your data:

Setting up and using Apache Fuseki provides practical experience with SPARQL and RDF data management, essential for creating and querying knowledge graphs. This skill is pivotal for AI applications that rely on semantic web technologies for data integration, reasoning, and complex querying capabilities. Efficiently harnessing these technologies enables more sophisticated data analysis and interaction within AI systems, directly contributing to their intelligence and effectiveness.

Task #2.2: Learning SPARQL with Fuseki

With your own SPARQL server set up, you're now equipped to explore queries in a controlled, personal environment, freeing you from the constraints and potential downtimes of external servers. Dive into the SPARQL Tutorial provided by Apache Jena, which includes a variety of datasets and guided query exercises tailored to reinforce your understanding of SPARQL's capabilities.

As you work through the tutorial, focus on understanding how different types of SPARQL queries (SELECT, ASK, CONSTRUCT, DESCRIBE) can be used to extract or manipulate data in diverse ways. Experiment with modifying the example queries to see how changes affect the results. Pay special attention to the use of filters, prefixes, and optional statements to refine your queries and retrieve more specific data.

Document your process, noting any challenges you encounter and how you overcome them. This reflection will not only solidify your understanding but also serve as a valuable resource for future projects. Feel free to explore beyond the tutorial's scope, testing queries on the datasets you uploaded to your Fuseki server. This hands-on experience is crucial for developing a nuanced understanding of how SPARQL interacts with knowledge graphs in real-world applications.

Task #3: SPARQL with Python

This task builds upon your existing knowledge of RDFlib, extending it with the capability to execute SPARQL queries directly within your Python applications. By integrating SPARQL queries with RDFlib, you'll gain a more flexible and powerful toolset for manipulating and querying RDF data programmatically. This approach not only streamlines the process of working with complex datasets but also opens up new possibilities for data analysis and application logic.

Task #3.1: RDFlib

Begin by deepening your understanding of querying RDF data with SPARQL through RDFlib. The Querying with SPARQL section of the RDFlib documentation will guide you through the syntax and capabilities of SPARQL within the RDFlib framework. This foundation is crucial for the tasks that follow.

Task #3.2: Query University Data

Now apply what you've learned by loading RDF triples (e.g., FOAF/FOCU data from last week) into your RDFlib-powered program. Experiment with SPARQL to perform queries such as listing students by age, finding information for a specific student, and aggregating students by university. This exercise will demonstrate the power of SPARQL in extracting and analyzing data from RDF graphs:

1. List all students, sorted by age
2. Find all predicates and objects for a given student, searching by first name (e.g., *"Joe"*)
3. Print a count of all students by university

Task #3.3: Smart University Agent v1.1

Refine the intelligent university agent you developed previously by incorporating SPARQL queries to interact with the graph. This modification aims to replace manual graph traversal with more efficient, query-based data retrieval. By doing so, you'll enhance the agent's functionality and efficiency, leveraging the full potential of SPARQL within a Python environment.

Further Experiments (try these on your own)

This task emphasizes the synergy between RDFlib and SPARQL, showcasing how they can be combined to create sophisticated data-driven applications. For querying *external* SPARQL endpoints, such as your own Apache Fuseki server, RDFlib integrates with SPARQLWrapper, a Python library designed to work with SPARQL endpoints. SPARQLWrapper simplifies executing SPARQL queries and managing responses, enabling your Python applications to interact with remote knowledge graphs effectively. Learn more about SPARQLWrapper by reading its documentation.

This linkage between local Python development and remote SPARQL services illustrates a powerful pattern for building scalable, intelligent systems that leverage semantic web data.

Conclusions

This lab has advanced your journey into the world of intelligent systems using knowledge graphs, building upon your knowledge of RDF and RDFS from previous labs. Through three focused tasks, you've explored the practical aspects of SPARQL, enhancing your skills in data querying and manipulation essential for intelligent systems development.

Recap

In **Task #1**, you engaged with DBpedia's public SPARQL endpoint (powered by OpenLink Virtuoso), to execute queries against a large-scale knowledge graph. This experience highlighted SPARQL's role in accessing globally distributed semantic data.

Note that Wikidata has a similar public endpoint with some additional features, which we will cover in a later lab.

Task #2 introduced you to setting up your own SPARQL server using Apache Fuseki, enabling personal experimentation and understanding of data hosting and querying mechanisms.

With **Task #3**, you applied SPARQL within a Python environment using RDFlib, merging programming with semantic web technologies to query and manipulate RDF data efficiently.

Together, these tasks have equipped you with the ability to design and develop intelligent systems that leverage semantic technologies. You're now prepared to tackle advanced challenges, applying your skills to create solutions that intelligently interact with, reason about, and utilize vast amounts of structured knowledge.

Outlook

Many leading technology companies and services employ knowledge graphs to enhance user experience and functionality. For instance, Google's search engine utilizes a vast knowledge graph to understand the relationships between entities and concepts, delivering precise search results and rich snippets. Its architecture, at a high level, involves querying structured data, much like you've practiced with SPARQL queries, to extract and present information relevant to user queries.

Similarly, LinkedIn uses a knowledge graph to map professional networks, skills, and job opportunities, enabling highly targeted recommendations and connections. This system relies on sophisticated querying of interconnected data, echoing the SPARQL-based manipulations and RDF data structuring you've learned.

These examples underscore the real-world applicability of semantic web technologies and knowledge graphs in creating complex, intelligent systems that can navigate and make sense of vast datasets, directly aligning with the skills and concepts covered in your lab exercises.

Armed with the knowledge from these labs, you're well-prepared to embark on smaller-scale, yet impactful, AI projects. For instance, you could develop a personalized recommendation system for books, movies, or music (recommender engines will be the topic of the next lecture). By constructing a knowledge graph of user preferences, item metadata, and their interrelations, and querying it with SPARQL, you can design algorithms that suggest items based on user behavior and preferences, similar to the underlying mechanisms of recommendation engines in platforms like Netflix or Spotify.

More concretely, in our course project, you will use knowledge graphs to build a smart educational assistant that helps students find resources related to their courses. By integrating course descriptions, educational resources, and student queries into a knowledge graph, you can create an application that uses SPARQL queries to provide personalized learning materials, leveraging RDF and SPARQL skills to filter and rank results based on relevance and user profiles.

These projects not only demonstrate the practical application of knowledge graph technologies and intelligent systems design on a manageable scale but also offer a glimpse into the process of developing applications that can understand and interact with complex datasets, mirroring the capabilities of larger AI systems in a more accessible context.

That's all for this lab!