



## WEB SEMANTIC COURSE

---

# Semantic Web Integration for CoopCycle : Data Collection and Querying in RDF

---

*Submitted To:*

Antoine Zimmerman

Victor Charpenay

*Submitted By :*

Saad Riffi Temsamani

Iness Bouabid

January 9, 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Goal of the project . . . . .	3
1.2	Coopcycle overview . . . . .	3
<b>2</b>	<b>System architecture</b>	<b>4</b>
2.1	Overview on Triplestore . . . . .	4
2.2	Highlight on the key components of the system . . . . .	4
2.2.1	”Collect” Component . . . . .	4
2.2.2	”Query” Component . . . . .	5
2.2.3	”Describe” Component . . . . .	5
<b>3</b>	<b>Data collection</b>	<b>6</b>
3.1	Html Parsing of the websites . . . . .	6
3.2	Structured Information Exposure through JSON-LD . . . . .	6
3.3	Integration with RDF and Triplestore . . . . .	7
3.4	SHACL for Rdf validation . . . . .	7
3.5	Checking the dataset . . . . .	7
<b>4</b>	<b>Querying</b>	<b>9</b>
4.1	SparQL queries . . . . .	9
4.1.1	Date and time query . . . . .	9
4.1.2	Location query . . . . .	10
4.1.3	Delivery price . . . . .	12
4.2	User preferences to enhance the dynamic use of the application . . . . .	13
4.2.1	Manual choices . . . . .	14
4.2.2	User preferences using a the website . . . . .	14

- 4.2.3 User preferences from different input types . . . . . 15
- 5 Describe 17**
  - 5.1 Purpose of the describe module . . . . . 17
- 6 Conclusion 18**

# Chapter 1

## Introduction

### 1.1 Goal of the project

In response to the evolving landscape of local business operations, this project introduces a Python-based software solution tailored for integration with CoopCycle. Using Semantic Web technologies like RDF and SPARQL, the application encompasses 3 core modules: a data collection program ("**collect**") that efficiently extracts and stores information from CoopCycle into a Triplestore, a querying interface ("**query**") that enables users to explore restaurants based on diverse criteria, and a user preferences configuration tool ("**describe**") allowing personalized RDF graph creation and utilizing Tuttle files. This report gives an overview of each module, highlighting the project's commitment to efficient data management and user-centric customization.

### 1.2 Coopcycle overview

CoopCycle serves as a collaborative platform connecting cooperatives with local businesses through dedicated websites. Each cooperative, often associated with a specific city, offers an array of restaurants across different locations. Utilizing **JSON-LD**, CoopCycle exposes detailed information about these businesses, including city, opening hours, delivery prices, and menus. To collect this valuable data, an HTML parser is essential, capable of efficiently navigating web pages and extracting JSON-LD information, ensuring a seamless and effective data collection process.

# Chapter 2

## System architecture

### 2.1 Overview on Triplestore

In our Semantic Web project, we use a triplestore, specifically Fuseki, as a central repository for managing RDF data. The triplestore stores information in the form of subject-predicate-object triples, facilitating the representation of the data.

As we collect data related to CoopCycle services, business descriptions, and user preferences in RDF format, the triplestore becomes an important component for storing and organizing this information. The RDF triples, enable us to capture complex relationships and represent entities such as restaurants, cooperatives, businesses, and user preferences.

Our workflow involves inserting collected RDF data into the triplestore, ensuring the integrity of the stored information. The triplestore acts as a powerful query engine, enabling us to perform semantic queries using SPARQL and uncover new information through inference. This capability is crucial for meeting the project's objectives, such as providing a food delivery discovery service based on user preferences, location, time, and price range.

### 2.2 Highlight on the key components of the system

#### 2.2.1 "Collect" Component

The core functionality of the collect function lies in collecting and structuring data for the project. Initially, it checks the triplestore's status and proceeds to fetch information from the

CoopCycle JSON API if the store is empty. This data, containing city details, CoopCycle URLs, and more, is transformed into the RDF format with predefined predicates and objects. Simultaneously, detailed restaurant data is scraped from CoopCycle URLs, converted to RDF, and inserted into the triplestore, ensuring data consistency. To maintain integrity, existing data is cleared before insertion. The resulting RDF graph is serialized into a Turtle file, to give us the foundation for subsequent semantic queries and meaningful exploration of data.

### **2.2.2 "Query" Component**

The query program is a command-line tool that helps users find restaurants based on specific criteria. It uses SPARQL queries to enable users to search for open restaurants at particular dates and times, explore options within specified zones or distances from a location, and discover restaurants that offer delivery below a set price. Users can also choose to rank results by either distance or minimum delivery price. The program's straightforward command-line interface makes it easy for users to get personalized restaurant recommendations based on their preferences.

### **2.2.3 "Describe" Component**

The "Describe" module serves as an interactive tool for users to set their restaurant preferences. By posing a series of straightforward questions, such as location, max\_price,...etc the program collects essential details. Subsequently, it transforms these inputs into a structured RDF graph, adhering to semantic web standards. The RDF graph is then published on a RDF oriented database platform, enabling easy storage and retrieval. This approach ensures that preferences are accurately captured, allowing users to save and reuse them for future restaurant searches. This modlue simplifies the process of personalizing restaurant preferences and interacts with resources.

# Chapter 3

## Data collection

### 3.1 Html Parsing of the websites

The HTML parsing component of the program demonstrates adaptability in handling diverse website structures. It extracts structured data, even when information is embedded within "application/json-ld" of a website. For specific details like restaurant URLs, the program dynamically navigates the HTML structure. The **JsonLDScraper** function accommodates scenarios where URLs are stored within(href) attributes inside (div) elements.

### 3.2 Structured Information Exposure through JSON-LD

The program extends to exposing structured information through JSON-LD. This format is crucial for capturing details about CoopCycle elements.

The JsonLDScraper function, in particular, plays an important role in this process. It scrapes restaurant URLs from the CoopCycle API and adeptly extracts structured data using the JSON-LD format. The program's ability to seamlessly integrate this information into the RDF graph ensures a comprehensive representation of CoopCycle elements. This structured information enhances the program's adaptability in capturing detailed data from CoopCycle websites.

### 3.3 Integration with RDF and Triplestore

The RDF and Triplestore components of the program utilize the *RDFlib* and *SPARQLWrapper* libraries to interact with Triplestore, specifically Apache Jena Fuseki.

The **import\_data** function initiates the retrieval of existing triplets from Triplestore, converting them into a *ConjunctiveGraph* for further processing. In case of errors during this process, appropriate error messages are handled. The **insert\_data** function facilitates the insertion of new data into Triplestore by executing an insert query. Importantly, it follows a comprehensive approach by first deleting existing data using the **delete\_data** function before the insertion, ensuring data accuracy.

For searching specific data, the **search\_data** function enables the execution of custom search queries on Triplestore. Lastly, the **visualize\_data** function provides a simple utility to print and visualize the retrieved data. Together, these functions establish a robust connection between the program and Triplestore, enabling seamless data integration, retrieval, and manipulation.

### 3.4 SHACL for Rdf validation

Before inserting coops and restaurants into Triplestore, a validation process is implemented. This validation specifically targets the RDF representations of cooperatives and restaurants, testing them against predefined constraints. This validation is done using an embedded module in python "*pyshacl*" which calls for the validation function which will take as a parameter a Turtle file taht contains constraitns. By doing so, the system verifies that the RDF data has the expected structure, relationships, and other criteria specified in the constraints. This validation acts as a filter, preventing the integration of malformed or inconsistent data, because, in some cases the entries are specified by the user and they may contain undefined values.

### 3.5 Checking the dataset

Upon completion of the collect program, the extracted data seamlessly integrates into the Triplestore, setting the stage for extensive information extraction through SPARQL queries. We can then, check if the collection is done succesfully by runnig a sparql test query in the Apache Jena interface as shown below in the images.



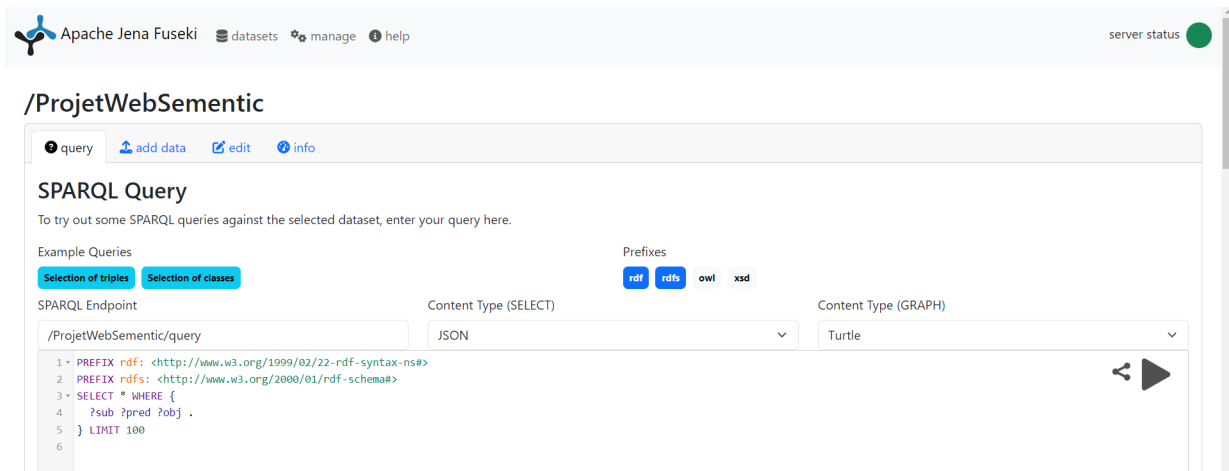


Figure 3.1: Apache Jena Interface

Table Response 100 results in 0.524 seconds Simple view Ellipse Filter query results Page size: 50

	sub	pred	obj
1	b0	<http://schema.org/priceSpecification>	b1
2	b0	<http://schema.org/deliveryMethod>	http://purl.org/goodrelations/v1#DeliveryModeOwnFleet
3	b0	<http://www.w3.org/1999/02/22-rdf-syntax-ns#>	<http://schema.org/OrderAction>
4	b0	<http://schema.org/target>	b2
5	b3	<http://www.w3.org/1999/02/22-rdf-syntax-ns#>	<http://schema.org/OpeningHoursSpecification>
6	b3	<http://schema.org/opens>	19:00
7	b3	<http://schema.org/closes>	21:30
8	b3	<http://schema.org/dayOfWeek>	Tuesday
9	b3	<http://schema.org/dayOfWeek>	Thursday
10	b3	<http://schema.org/dayOfWeek>	Saturday
11	b3	<http://schema.org/dayOfWeek>	Friday
12	b3	<http://schema.org/dayOfWeek>	Wednesday
13	<https://kooglof.coopcycle.org/api/restaurants/7...>	<http://schema.org/openingHoursSpecification>	b4
14	<https://kooglof.coopcycle.org/api/restaurants/7...>	<http://schema.org/openingHoursSpecification>	b5
15	<https://kooglof.coopcycle.org/api/restaurants/7...>	<http://schema.org/openingHoursSpecification>	b6
16	<https://kooglof.coopcycle.org/api/restaurants/7...>	<http://schema.org/potentialAction>	b7
17	<https://kooglof.coopcycle.org/api/restaurants/7...>	<http://schema.org/image>	<https://kooglof.coopcycle.org/media/cache/restaurant_thumbnail/65/6e/656e168020d14.png>
18	<https://kooglof.coopcycle.org/api/restaurants/7...>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#>	<http://schema.org/Restaurant>
19	<https://kooglof.coopcycle.org/api/restaurants/7...>	<http://schema.org/sameAs>	<https://donation-maitre-eclair.fr/>
20	<https://kooglof.coopcycle.org/api/restaurants/7...>	<http://schema.org/name>	Donation Maitre Eclair 🍩

Figure 3.2: Result of SELECT \* query

# Chapter 4

## Querying

### 4.1 SparQL queries

The initial phase in extracting data from the triplestore involves crafting well-formed SPARQL queries. The query module within our system serves as a crucial component for dynamically generating queries based on both input parameters and user preferences. This module offers functions like **search\_by\_place()**, **get\_by\_max\_price()**, and **generate\_search\_query()**, each designed to make queries to specific search criteria.

Once the query is dynamically generated, it is advantageous to inspect and verify its correctness before executing it. To facilitate this verification process, the system includes a feature that enables the printed display of the final SPARQL query in the terminal. This visibility allows users to review the query structure, ensuring it aligns with their expectations. It also provides for users to copy and paste the query into Apache Jena interface for manual inspection and validation.

#### 4.1.1 Date and time query

The **SELECT** clause specifies the variables to be returned in the results: `restaurant_link`, `name`, `openingTime`, `closingTime`, and `address`. These variables capture information about the restaurants.

The **FILTER** clause ensures that only restaurants open during the specified timeframe on the specified day are included in the results.

```
def generate_search_query(date, time):
    search_query = "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>\n"
    search_query += "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>\n"
    search_query += "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n"
    search_query += "PREFIX pwp:<https://ProjectW9s.com/predicate/>\n"
    search_query += "PREFIX pwo:<https://ProjectW9s.com/object/>\n"
    search_query += "PREFIX pws:<https://ProjectW9s.com/subject/>\n"
    search_query += "PREFIX schema: <http://schema.org/>\n"
    search_query += "SELECT ?restaurant_link ?name ?openingTime ?closingTime ?address\n"
    search_query += "WHERE {\n"
    search_query += "?restaurant a schema:Restaurant ;\n"
    search_query += "schema:restaurant ?restaurant_link ;\n"
    search_query += "schema:name ?name;\n"
    search_query += "schema:address ?address_link;\n"
    search_query += "schema:openingHoursSpecification [\n"
    search_query += "schema:opens ?openingTime ;\n"
    search_query += "schema:closes ?closingTime ;\n"
    search_query += "schema:dayOfWeek ?dayOfWeek\n"
    search_query += "]\n"
    search_query += "?address_link a schema:PostalAddress;\n"
    search_query += "schema:streetAddress ?address.\n"

    search_query += f"FILTER (?dayOfWeek = \"{date}\" && ?openingTime <= \"{time}\" && ?closingTime > \"{time}\")\n"
    search_query += "}"

    return search_query
```

Figure 4.1: Result of SELECT \* query

### 4.1.2 Location query

The location querying program consists of two primary components, each serving a distinct purpose. In the first part, the program interacts with users to collect their location criteria, offering three distinct options:

- City name.
- Latitude and longitude of a place.
- Distance from a specific place.

Regardless of the chosen method, the collected criteria are stored in a dictionary that includes the information type, coordinates, and a maximum distance variable.

#### Location criteria dictionary

In order to get the correct information to send to the query, the program relies on the **Geopy** library, which provides *geocoding* and *reverse geocoding* functionality. The **Nominatim** class from Geopy is utilized to convert place names like cities into latitude and longitude coordinates, ensuring accurate location data for user input.

The program also utilizes the **geodesic** class from Geopy to calculate distances between coor-

dinates accurately. This is crucial for scenarios where the user chooses to search for restaurants based on proximity, either by city, coordinates, or distance from a specific place. The geodesic class enables the program to measure distances in kilometers, providing essential functionality for filtering restaurant results within a specified radius.

Once the location criteria dictionary is created, it becomes the foundation for constructing the SPARQL queries tailored to the users' preferences.

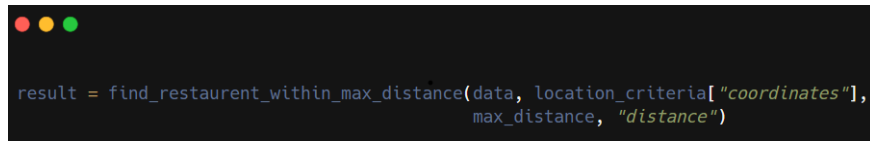
Below, is an exemple of how a query is generated based on the location criteria dictionary.

```
def search_by_place(day, time, type, coordinates):
    search_query = (
        "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>\n"
        "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>\n"
        "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n"
        "PREFIX pwp:<https://ProjectW9s.com/predicate/>\n"
        "PREFIX pwo:<https://ProjectW9s.com/object/>\n"
        "PREFIX pws:<https://ProjectW9s.com/subject/>\n"
        "PREFIX schema: <http://schema.org/>\n"
    )

    if type == "address" or type == "geocord":
        search_query += (
            "SELECT ?restaurant_link ?name ?openingTime ?closingTime ?address ?latitude ?longitude\n"
            "WHERE {\n"
            "?restaurant a schema:Restaurant ;\n"
            "schema:restaurant ?restaurant_link ;\n"
            "schema:name ?name;\n"
            "schema:address ?address_link;\n"
            "schema:openingHoursSpecification [\n"
            "schema:opens ?openingTime ;\n"
            "schema:closes ?closingTime ;\n"
            "schema:dayOfWeek ?dayOfWeek\n"
            "]\n"
            "?address_link a schema:PostalAddress;\n"
            "schema:streetAddress ?address.\n"
            "?address_link a schema:PostalAddress;\n"
            "schema:geo ?coordinates.\n"
            "?coordinates a schema:GeoCoordinates;\n"
            "schema:longitude ?longitude ;\n"
            "schema:latitude ?latitude.\n"
            f"FILTER (?dayOfWeek = \"{day}\" && ?openingTime <= \"{time}\" && ?closingTime > \"{time}\")\n"
            "}"
        )
```

Figure 4.2: Exemple of a generated query for searching by location

After creating the SPARQL query based on the user's location criteria, the program uses the **search\_data()** function to execute the query against the triplestore. The result, stored in a variable, holds information about CoopCycle restaurants meeting the specified conditions. This variable becomes crucial for subsequent processing, including sorting and visualization, enabling tailored restaurant recommendations for the user.



```
result = find_restaurant_within_max_distance(data, location_criteria["coordinates"],
                                             max_distance, "distance")
```

Figure 4.3: Searching by location and using the max distance for delivery service

### 4.1.3 Delivery price

In this code snippet, a second query function, **get\_by\_max\_price**, has been implemented to retrieve CoopCycle restaurants that accept delivery orders below a specified maximum price. This query searches for restaurants based on the provided day, time, and maximum delivery cost criteria. The SPARQL query includes conditions to filter restaurants meeting the specified criteria, such as opening and closing times, day of the week, and delivery cost. The result set includes relevant information such as the restaurant's name, address, opening and closing times, and delivery cost.

The function **get\_restaurants\_by\_max\_delivery\_price** utilizes this query function, allowing users to find restaurants within a certain distance from their location that meet the specified maximum delivery price.

The program prompts users for input related to location criteria, and the generated SPARQL query is executed using the **search\_data** function.

The generated query is the output of the function `get_by_max_price` :

```
def get_by_max_price(day, time, max_price=None):
    query = """
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema>
PREFIX pwp: <https://projectw9s.com/predicate/>
PREFIX pwo: <https://projectw9s.com/object/>
PREFIX pws: <https://projectw9s.com/subject/>
PREFIX schema: <http://schema.org/>

SELECT ?restaurant_link ?name ?openingTime ?closingTime ?address ?delivery_cost ?latitude ?
longitude
WHERE {
    ?restaurant a schema:Restaurant ;
    schema:restaurant ?restaurant_link ;
    schema:name ?name;
    schema:address ?address_link;
    schema:openingHoursSpecification [
        schema:opens ?openingTime ;
        schema:closes ?closingTime ;
        schema:dayOfWeek ?dayOfWeek;
    ];
    schema:potentialAction[
        a schema:OrderAction;
        schema:priceSpecification[
            a schema:DeliveryChargeSpecification;
            schema:eligibleTransactionVolume[
                a schema:PriceSpecification;
                schema:price ?delivery_cost ]]].

    ?address_link a schema:PostalAddress;
    schema:streetAddress ?address.
    ?address_link a schema:PostalAddress;
    schema:geo ?coordinates .
    ?coordinates a schema:GeoCoordinates;
    schema:longitude ?longitude ;
    schema:latitude ?latitude .

    FILTER(?dayOfWeek="%s" && ?openingTime <= "%s" && ?closingTime > "%s" && ?delivery_cost <= %f)
}

ORDER BY ?delivery_cost
return go(f, seed, [])
}
```

Figure 4.4: Generating search query for sorting by max price of delivery

## 4.2 User preferences to enhance the dynamic use of the application

The focus is on diversifying the methods for obtaining user preferences to enhance the application's dynamic functionality. The implemented functionality allows users to provide preferences through various means such as a website link, a Turtle file, or an RDF graph. This versatility empowers the application to adapt to different user scenarios, providing a more interactive and user-friendly experience.

### 4.2.1 Manual choices

The key feature discussed in this section revolves around a function that leverages the **–rank-by** argument, allowing users to specify whether they want to rank search results based on distance or price. The function then utilizes existing functions, such as the get restaurants by max delivery price for ranking by price and get restaurants by place for ranking by distance. These functions, integrated into the application, provide a dynamic and responsive approach to user preferences, enabling the application to tailor restaurant searches based on the user’s desired ranking criteria.

The user can then interact with the application using the following command, in this exemple the rankin is done using the max price function:



Figure 4.5: Command line for ranking by price

In this phase, the program prompts users for specific details, like the maximum price or preferred location, using one of the three methods discussed earlier in the query-by-location section. This manual interaction involves users making choices through the program’s interface.

### 4.2.2 User preferences using a the website

The subsequent section introduces another command line argument, **–user-preferences**, offering the option to load user preferences from an RDF file. This command facilitates a more streamlined and automated process for users with predefined preferences, enhancing program flexibility to cater to both manual, interactive inputs and automated loading of preferences from RDF files. Assuming that the source for user preferences is already initialized with the URI :

<https://www.emse.fr/zimmermann/Teaching/SemWeb/Project/pref-charpenay.ttl>

The user then creates a command line as follows :

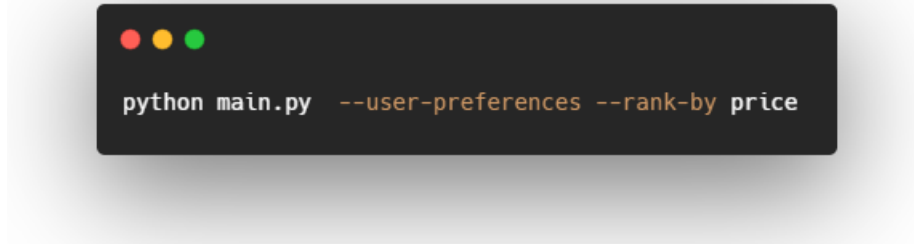


Figure 4.6: Command line for user preferences from the link ranking by the price

This approach involves extracting specific content from the Turtle file provided via the link, setting these details as user preferences for ranking. With this method, users are relieved from manually entering the maximum price or specifying their preferred location. Instead, the program automatically retrieves the relevant information from the linked Turtle file, streamlining the user experience and enhancing convenience.

### 4.2.3 User preferences from different input types

In this advanced user preferences extraction section, users can opt to retrieve their preferences in multiple ways:

- By providing a link to a website hosting a Turtle file.
- By uploading a local Turtle file containing RDF.
- By using a graph.

The system is designed to handle various user inputs seamlessly.

To facilitate this, a dedicated dataset for user preferences is introduced, and we can access to it using Fuseki Apache Jena through the link **localhost:3030/Users** allowing users to store and manage their preferences efficiently.

A program is implemented to query this dataset based on a user's username, enabling the retrieval of preferences stored in a graph. If a user hasn't set any preferences yet, the system employs the "import and insert" function to add preferences to the triplestore. Subsequently, users can effortlessly extract their preferences, offering flexibility for creating and managing



SPARQL Endpoint

Content Type (SELECT)

Content Type (GRAPH)

/Users/

JSON

Turtle

```

1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 SELECT * WHERE {
4   ?sub ?pred ?obj .
5 }

```

Table

Response

18 results in 0.018 seconds

Simple view

Ellipse

Filter query results

Page size: 50

sub	pred	obj
1 <https://projectw9s.com/users/Saad>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://schema.org/Person>
2 <https://projectw9s.com/users/Saad>	<http://schema.org/name>	Saad
3 <https://projectw9s.com/users/Saad>	<http://schema.org/address>	b0
4 <https://projectw9s.com/users/Saad>	<http://schema.org/seek>	b1
5 b0	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://schema.org/PostalAddress>
6 b0	<http://schema.org/postalCode>	42100
7 b0	<http://schema.org/addressLocality>	Saint-Étienne
8 b1	<http://schema.org/priceSpecification>	b2
9 b1	<http://schema.org/availableAtOrFrom>	b3
10 b1	<http://schema.org/ranking>	price
11 b2	<http://schema.org/maxPrice>	"15.0"^^<http://www.w3.org/2001/XMLSchema#decimal>
12 b2	<http://schema.org/priceCurrency>	EUR
13 b3	<http://schema.org/geoWithin>	b4
14 b4	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://schema.org/GeoCircle>
15 b4	<http://schema.org/geoMidpoint>	b5
16 b4	<http://schema.org/geoRadius>	"2000.0"^^<http://www.w3.org/2001/XMLSchema#decimal>
17 b5	<http://schema.org/longitude>	"4.3873058"^^<http://www.w3.org/2001/XMLSchema#decimal>
18 b5	<http://schema.org/latitude>	"45.4401467"^^<http://www.w3.org/2001/XMLSchema#decimal>

Figure 4.7: Users dataset

multiple preference profiles.

# Chapter 5

## Describe

### 5.1 Purpose of the describe module

In the application's interactive introduction, users are welcomed and provided with a choice to initiate a search or exit the application. The program is designed to guide users through these options, ensuring a user-friendly experience. Upon selecting the search option, users are prompted to input various preferences such as their full name, address, postal code, maximum distance, and maximum price for orders. Additionally, users can choose their preferred ranking criterion – either by distance or price.

For those wishing to save their preferences for future use, the system offers a user registration option. Users are prompted to decide whether they want to store their preferences, and if affirmative, they input the necessary details. The registration process ensures unique names and valid locations, enhancing the reliability of the stored preferences. A user profile is then created and saved, allowing for subsequent personalized searches.

The **usage\_profile()** function manages the user's interaction with preferences. Users can check if they have saved preferences before, and if not, they are guided through the registration process. Existing users can retrieve their preferences by entering their registered name. The system verifies the user's identity and extracts their preferences, facilitating a seamless and personalized experience.

# Chapter 6

## Conclusion

Leveraging semantic web technologies, geospatial data, and dynamic user preferences, the application offers a comprehensive solution for personalized restaurant searches. Users can input their location criteria through city names, coordinates, or proximity to a specific place, while also specifying preferences through various methods.

The system's modular architecture ensures scalability and adaptability, accommodating diverse user needs. The integration of geolocation functionality adds a valuable layer to the search process, allowing users to find restaurants based on proximity. Additionally, the application provides flexibility in obtaining user preferences, whether through links, Turtle files, or direct graph interaction. The usage scenarios and user interaction elements further enhance the accessibility and user-friendliness of the application.