Representing Jupyter Notebooks with Knowledge Graphs to Address Data Lineage Problems

Auteur : Birtles, Alixia
Promoteur(s) : Debruyne, Christophe
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en science des données, à finalité spécialisée
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/20479

UNIVERSITY OF LIÈGE

SCHOOL OF ENGINEERING

Representing Jupyter Notebooks with Knowledge Graphs to Address Data Lineage Problems

Master Thesis carried out to obtain the degree of Master in Data Science and Engineering

Author
BIRTLES Alixia

Supervisor
DEBRUYNE Christophe

Academic year 2023-2024

Abstract

In data science, data lineage is a crucial aspect that is often insufficiently considered. To address challenges related to data lineage, the approach presented in this thesis leverages knowledge graphs and data provenance.

The PROV-O ontology and the FOAF vocabulary are harnessed to design a structure, along with defined terms. This ontology aims to represent the information extracted from Jupyter notebooks, tools often used in data science. Additionally, public APIs are leveraged to enrich the graph.

Initially, the RML language was used to map the data, but it was too limiting and led to the consideration of the RDFLib library in Python. RMLMapper and Morph-KGC have been considered, but the former does not have the required extension to access the desired data in the source code, while the latter has iterator challenges and does not support theta-joins.

The correctness of the approach was validated with visualization in GraphDb and SPARQL queries. A complex query related to the extraction of licenses demonstrated the feasibility of the approach and the ability to answer questions about data lineage. Moreover, experimentation with queries on a real-world dataset, the KGTorrent dataset, showed the effectiveness of the approach. Performance measurements on the construction of the graph and on SPARQL queries in real-world conditions led to promising results.

Contents

Acknowledgements

I would like to express my warmest thanks to my supervisor, Christophe Debruyne, who made this work possible. His guidance and advice carried me through all the stages of writing my master's thesis. This work is the result of his continuous support, his insightful feedback, and his patience. His interest in my thesis and his availability have been invaluable. He also gave me the opportunity to present my work during the FWO Scientific Research Network Knowledge Graphs for Data Integration seminar organized at the University of Liège, which was a rewarding experience.

I am also deeply thankful to my parents and my siblings as a whole for their continuous support and understanding when undertaking my academic journey and achieving my master thesis. Their belief in me and their encouragement during challenging times have been a constant source of strength. Specifically, I would like to thank my mother, Cécile Moitroux, for her unwavering support. She has helped me overcome difficulties and motivated me to pursue my goals. She let me become the best version of myself.

Finally, I would like to thank my friends for their support and encouragement. They have been a source of liberty and joy throughout my studies. Their kindness has been a source of inspiration.

Their contributions have profoundly influenced this thesis, and I am truly grateful to them for their support.

Chapter 1

Introduction

1.1 Context

Data science is a rapidly growing field that plays an increasingly important role in technology. This field is related to extracting knowledge from data, which has been experiencing growth in volume, variety, and velocity in the last few years. This data can be represented in different formats, such as structured, semi-structured, and unstructured. This diversity induces the need for a robust data management system to store, process, and analyze the data. One critical aspect of data management is understanding *data lineage*, which is the ability to trace the origin of data and their movements: transformations and dependencies. According to Robert Ikeda et al., "Lineage, or provenance, in its most general form describes where data came from, how it was derived, and how it was updated over time" [1]. This lineage, often related to *data provenance*, provides a way to understand how data evolves and ensure its reproducibility and reliability.

Nowadays, the use of Jupyter notebooks is significant in the field of data science. Jupyter notebooks are Web-based interactive computing platforms that allow users to create and share documents that contain code, equations, visualizations and narrative text. These notebooks are stored in JSON, which is a semi-structured data format that is easy for humans to read and write [2]. Metadata is present, within the JSON format of notebooks, regarding the structure of a notebook, the execution order of cells, and the source code, all of which can be extracted and exploited. This metadata also includes implicit information, such as the relationships between cells or notebooks. In the context of this thesis, this implicit information is transformed into an explicit representation.

The provenance of the data in Jupyter notebooks can be used to answer some questions related to a project developed using this platform. This data might help users reproduce the results of an experiment or analysis, understand the transformation process of the data along a pipeline, and be aware of the documentation of data transformations within a Jupyter notebook. Given that Jupyter notebooks are not always associated with high code quality standards, ensuring data lineage and reproducibility of results poses a significant challenge [3].

1.2 Problem statement

The field of data science presents significant challenges in terms of data provenance management, more specifically data lineage, and reproducibility [4, 5]. Data lineage refers to the ability to trace the origin of data and its movement across various systems, while reproducibility entails the capability to replicate the results of experiments or analyses reliably.

Considering these challenges, some main issues related to data lineage management can be identified. The re-execution of workflow does not reuse previous provenance data, which can lead to issues related to the storage or the updates. The sharing of data provenance suffers from a lack of clear annotations and explanations in the shared provenance data, making its interpretation difficult for other users or researchers. In more detail, specific challenges in data lineage management include data redundancies, difficulties in tracing transformations, and dependencies between data [4, 6]. These challenges highlight the critical importance of developing more effective approaches to dealing with data provenance to ensure transparency, reproducibility, and quality of research results in data science.

Therefore, when it comes to understanding challenges related to data lineage, numerous questions naturally arise, such as :

Q1 *What quality assurance measures have been applied to the data?*

Q2 *What is the transformation process of the data along a pipeline and how the transformations are documented?*

Q3 *What are the licenses or restrictions associated with the datasets used to train a given AI model, and how are these pieces of information are documented?*

Addressing these challenges can have several positive impacts. It might help to obtain a better comprehension of decision-making processes and provide confidence in the results of an analysis. Indeed, we would have a further understanding of the data. Moreover, improved data lineage management can lead to more efficient data sharing and the reuse of valuable datasets, which can be beneficial for the scientific community as a whole.

We must recognize the widespread use of Jupyter notebooks in data science. The provenance data in Jupyter notebooks is implicitly stored in its JSON format serialization, although this may vary depending on use cases or user preferences. This data can be leveraged to address challenges related to data lineage. The provenance data in Jupyter notebooks opens up new opportunities to explore innovative approaches to solve data lineage challenges in data science. We believe that extracting, structuring, and storing such provenance data in a *knowledge graph* with explicit semantics provides a feasible approach to better comprehending the aforementioned transformation processes, the data origins, and dependencies within a Jupyter notebook. Considering the discussion on the importance of data lineage management and the potential of leveraging provenance data and knowledge graphs, it naturally leads us to ask: **How can provenance information and knowledge graphs be leveraged to address data lineage challenges in data science?**

1.3 Objectives

The main purposes of this thesis are to extract the provenance of the data in Jupyter notebooks and to represent it as a knowledge graph to answer some questions related to the challenges of data lineage in data science. The knowledge graph is used to represent the provenance data in a structured way by adopting an extension of the *PROV-O ontology*. This ontology provides a standard and generic framework to represent provenance information related to interactions between entities, activities, and agents in different applications and domains [7].

To achieve this, there are some specific objectives to meet:

1. **Literature Review and Understanding of Challenges:** Review the literature on data lineage and provenance management in the field of data science to gain a comprehensive understanding of the data lineage in this context.

2. **Knowledge Graph Construction:** Investigate methods to represent the structure of the knowledge graph using an extension of the PROV-O ontology. Then, identify the most suitable language to design the mapping rules between the provenance data, extracted from Jupyter notebooks, and the knowledge graph.

3. **Validation Analysis of Knowledge Graph:** Analyze the correctness of the constructed knowledge graph through a visualization tool and SPARQL queries [8].

4. **Queries and Performance Measurements:** Utilize queries to get an overview of the kinds of questions related to the data lineage that can be answered. Additionally, build the knowledge graph in real-world conditions, query it, and analyze the performance measurements and the obtained results.

1.4 Outline

To ensure a clear and structured work for this thesis, it is essential to present a well-defined plan. This section outlines its structure, providing a detailed description of each chapter.

- **Chapter 1: Introduction**
  This chapter presents the context and the problem to be addressed, as well as the objectives to address the central research question.

- **Chapter 2: Background**
  The background chapter provides an overview of the concepts related to the research question and the method employed. It covers fundamentals concepts, such as provenance data, knowledge graphs, and ontologies. By providing a brief introduction to these concepts, this chapter lays the foundation for the subsequent chapters.

- **Chapter 3: Related Work**
  The related work chapter examines the literature on data lineage and provenance management in the field of data science. This chapter provides an overview of what has been done in the field and the challenges that have been addressed or those that remain to be addressed.

- **Chapter 4: Knowledge Graph Design**
  The design chapter describes the data sources from which the information is extracted and the structure of the knowledge graph. It also approaches the extension of the PROV-O ontology to address specific needs in this work.

- **Chapter 5: Implementation**
  In the implementation chapter, the language used to map the provenance data from Jupyter Notebooks to the structure of the knowledge graph is described. It also includes the limitations encountered during the implementation process and the solutions to overcome them.

- **Chapter 6: Evaluation and Experimentation**
  This chapter aims to provide a visualization and demonstration of queries with test notebooks. Additionally, it includes experimentation on a real-world dataset with performance measurements on the construction of the knowledge graph and on queries.

- **Chapter 7: Discussions**
  This chapter discusses the implications of the results and the limitations faced during the implementation and evaluation of the knowledge graph.

- **Chapter 8: Conclusions**
  This chapter summarizes the main findings of the thesis and provides recommendations for future work in the fields of data lineage and provenance management.

Chapter 2

Background

This chapter aims to familiarize readers with the fundamental concepts and tools that are used in the subsequent chapters. Furthermore, we highlight the importance and relevance of the following theoretical aspects in the context of this thesis. This aspires to aid readers in understanding specific concepts detailed and covered throughout the research, and to provide a solid foundation for future discussions and analyses.

As part of the research question, which aims to take the advantages of knowledge graphs to address challenges related to data lineage, it is essential to understand the fundamentals of the graph data model. This model provides an intuitive and flexible representation of relationships between data, serving as a robust foundation for constructing knowledge graphs. Moreover, it facilitates easy visualization, making it a versatile tool for data analysis and interpretation.

2.1 Graph Data Model

Data can be represented in various ways. The one we are interested in is the directed graph data model. In this model, data is represented as a graph composed of nodes and edges, where nodes represent entities and edges represent relationships between entities. The edges are directed, meaning that they have a direction from one node to another.

An example of a directed graph data model is illustrated in Figure 2.1. In this example, nodes are people or flowers, while edges represent relationships between them.

Directed graph data models offer an intuitive way to represent entities and the relationships between them, but how these graphs are formalized and exchanged between systems is another question. This is where the *Resource Description Framework* (RDF) comes into play. In the following section, we discuss RDF and its importance in the context of this thesis.

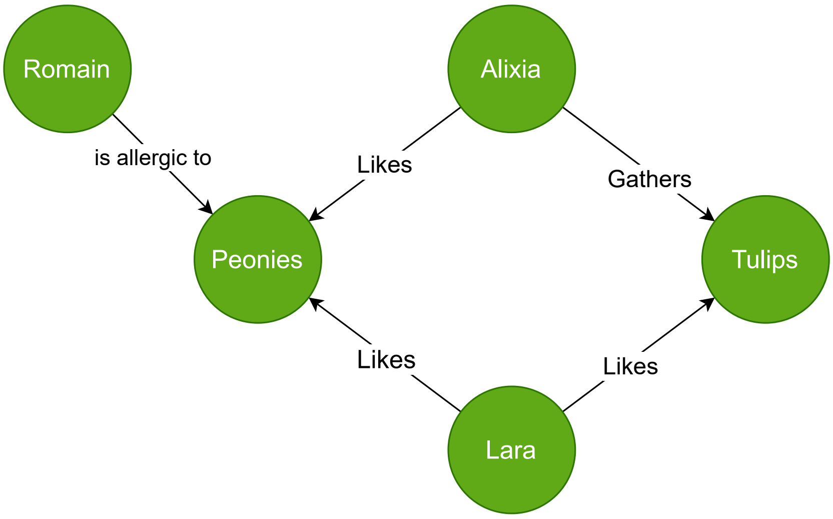


Figure 2.1: An example of a directed graph data model.

2.2 RDF

RDF stands for Resource Description Framework. Standardized by the World Wide Web Consortium (W3C), RDF is a data model for describing and exchanging resources on the Web in a structured and interoperable way. Interoperability is a key aspect of RDF, as it allows data to be shared and integrated across different systems and applications.

In RDF, resources are described using triples, which is a statement that consists of three parts: a subject, a predicate, and an object.

```
<subject> <predicate> <object>
```

The statement aims to formulate a relationship between two resources: the **subject** and the **object**, with the **predicate** defining the nature of their relationship.
As an example, consider the following triple: "Lara likes Tulips". `Lara` is the subject, `Tulips` is the object, and `likes` is the predicate that represents the relationship between `Lara` and `Tulips`.

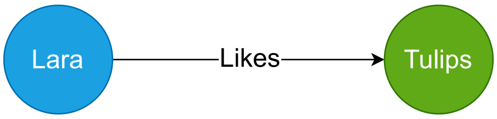


Figure 2.2: An example of an RDF triple.

To uniquely identify resources, RDF uses Internationalized Resource Identifiers (IRIs), which can be used as a subject, object, or predicate in a triple. In addition, literals are used to represent values such as strings, numbers, and dates, with only the object in the triple that can be a literal. The notion of a blank node is used to represent a resource that does not have a unique identifier to represent it; it is only assigned a local identifier [9]. Resume the previous example with the use of IRIs:

Subject: `<http://example.org/Person/Lara>`
Predicate: `<http://example.org/likes>`
Object: `<http://example.org/Flower/Tulips>`

In this example, the IRIs `http://example.org/Person/Lara` and `http://example.org/Flower/Tulips` represent the resources `Lara` and `Tulips`, respectively, and the IRI `http://example.org/likes` represents the relationship between them. This demonstrates how IRIs can uniquely identify resources in RDF triples, enabling the representation of relationships within data structures. As these relationships are interconnected, RDF statements form a graph structure.

Statements in RDF can be represented as a graph when they are connected by their relationships. An RDF graph is thus a set of RDF triples that represent a network of relationships, allowing the representation of knowledge in a structured way. This characteristic leads to RDF being described as a "Directed label graph data format for representing information on the Web" [9].

In particular, a collection of RDF graphs defines an RDF dataset. In an RDF dataset, each graph is identified by a unique IRI or a blank node, except for one, which is considered the default graph and does not have a name, represented by an IRI [10]. An example of an RDF dataset illustrated in Listing 2.1 is composed of two named graphs: g_flowers and g_preferences, and a default graph. The g_flowers graph contains information about flowers, while the g_preferences graph contains information about preferences of people. The last triple in the dataset is part of the default graph as it is not associated with a named graph.

```
@prefix ex: <http://example.org/> .
@prefix flower: <http://example.org/flowers#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

# Graph about flowers
ex:g_flowers {
    ex:Tulip rdf:type flower:Flower ;
            flower:color "red" ;
            flower:height "30cm" .
    ex:Rose rdf:type flower:Flower ;
            flower:color "pink" ;
            flower:height "40cm" .
}

# Graph about preferences
ex:g_preferences {
    ex:Lara rdf:type ex:Person ;
            ex:likes ex:Tulip .
```

```
        ex:John rdf:type ex:Person ;
                ex:likes ex:Rose .
    }

    # Default graph
    ex:Sunflower rdf:type flower:Flower ;
                 flower:color "yellow" ;
                 flower:height "50cm" .
```

Listing 2.1: An example of an RDF dataset.

A visualization of the RDF dataset represented in turtle format in Listing 2.1 is illustrated in Figure 2.3.

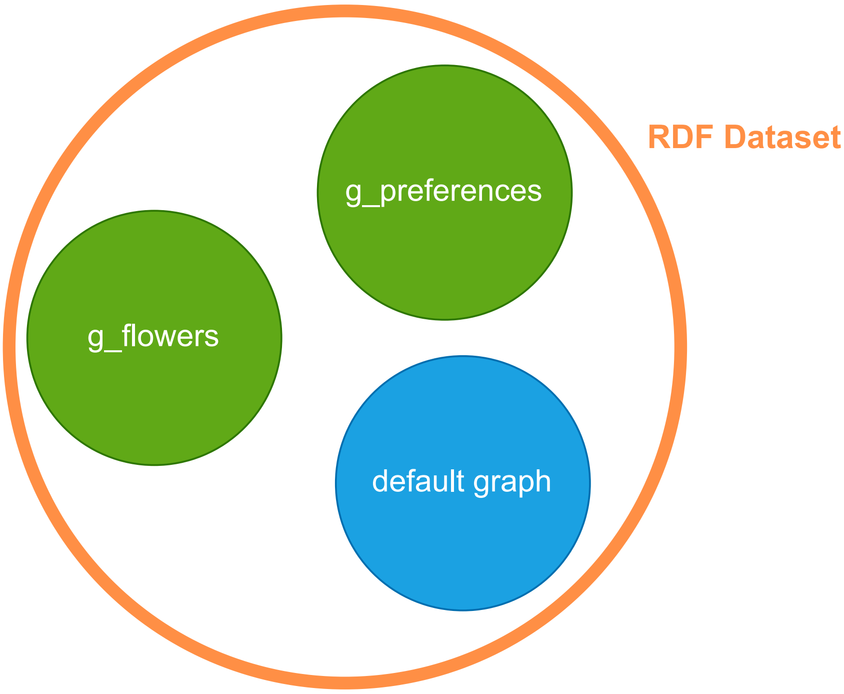


Figure 2.3: An example of an RDF dataset.

RDF data can be serialized in different formats, each with its own advantages and use cases. The one we use for this thesis is the Turtle format, which is a human-readable format that allows the representation of RDF data in a more compact way. In addition to Turtle, there are several other serialization formats, such as JSON-LD, N-Triples, and RDF/XML [9].

2.2.1 Relevance of RDF

RDF plays a crucial role in this thesis by providing a flexible and standard framework to represent and share data on the semantic Web. By adopting RDF, this work benefits from a standardized method to represent complex relations between entities and concepts. This enables a transparent integration of data from heterogeneous sources, facilitating the discovery of new knowledge. Furthermore, RDF offers inference capabilities that allow the extraction of implicit knowledge from explicit data, enhancing the power of the analysis drawn from this thesis.

2.3 Ontologies

An ontology can be viewed as a formal representation of knowledge in a specific domain, defining concepts and relationships between them in a structured way. The idea behind the creation of ontologies is to provide a common understanding of a domain, enabling knowledge sharing, reuse, and interoperability between systems and applications.

Ontologies are typically stored in a machine-readable format, often as Resource Description Framework, which allows for easy integration with other semantic web technologies. They offer various benefits, such as the ability to structure knowledge, to provide a common vocabulary, and to enable automated reasoning and inference about data. They can also be easily extended to meet specific requirements by adding new concepts and relationships. It is therefore an adaptable and flexible tool that evolves depending on the data.

For example, let us focuses on the class Person by considering an ontology in the context of genealogy. A genealogy ontology can be used to describe relationships between people, including parents, children, and siblings. It can also define properties such as `hasFather` and `hasMother` to describe the relationships between a child and their parents. The hierarchy of classes is illustrated in Figure 2.4. This ontology describes a family tree with the following classes: `Person`, `Father`, `Mother`, and `Child`. The `Person` class is the parent class of the `Father`, `Mother` and `Child` classes. The relationships between classes are defined by properties such as `hasFather` and `hasMother` and are illustrated in Figure 2.5.

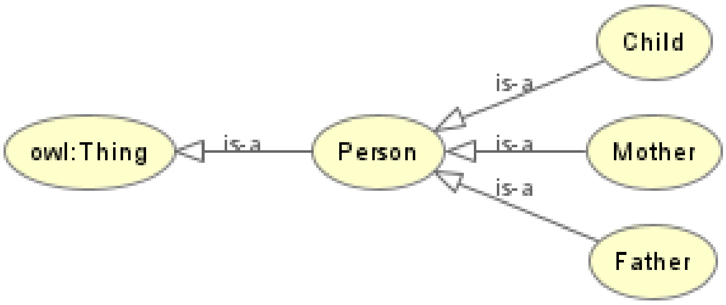


Figure 2.4: An example of a genealogy ontology: Structure of the classes.

There are different types of ontology languages; among them are *Resource description Framework Schema* (RDFS) and the *Web Ontology Language* (OWL). OWL is a Semantic Web language that allows the creation of ontologies with rich and complex knowledge, as RDFS is limited in its expressive power. OWL allows the description of resources and their relationships in a more detailed way, enabling the creation of more complex ontologies [11].

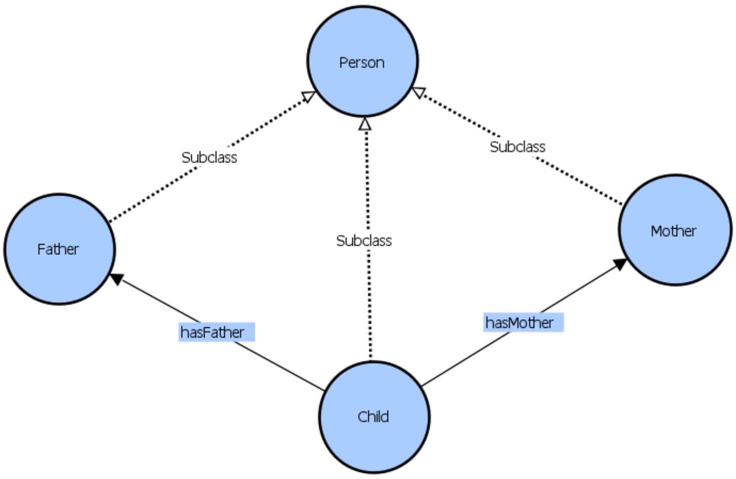


Figure 2.5: An example of a genealogy ontology: Relationships between classes.

2.3.1 Vocabularies

The notion of vocabulary is different from that of ontology. A vocabulary is a set of terms used to describe resources within a specific domain. It is used as a foundational element to build ontologies, providing a list of classes and properties to describe resources and relationships. We can view the vocabulary as the terminology that defines the concepts, while the ontology formalizes these concepts and their relationships.

For instance, the FOAF (Friend of a Friend) vocabulary is a set of terms used to describe people and their relationships, which includes terms like `foaf:Person`, `foaf:name`, and `foaf:knows` [12]. While the FOAF vocabulary describes basic concepts for individuals, an ontology can be built upon it to define more complex relationships by including constraints and axioms to describe the domain more precisely.

In the context of this thesis, we will use ontologies and vocabularies to represent knowledge in a structured way, enabling the creation of knowledge graphs to address data lineage challenges.

2.4 Knowledge Graphs

A knowledge graph is a graph that meets specific criteria. We have already seen in Section 2.1 the concept of a graph data model, which is a way to represent data as a graph composed of nodes and edges. The concept of knowledge graphs is to represent knowledge with this kind of graph. A knowledge graph needs to meet certain criteria to be considered as such. These criteria are the following:

- Ontologies are used to define the schema of the graph: the concepts and relationships between them.

- The graph is enriched with data from various domains, organizations, and sources.
- There is a support for reasoning and inference to derive new knowledge from existing data.

If those criteria are met, the graph can be considered as a knowledge graph. Otherwise it is just a graph. Therefore, all knowledge graphs are graphs, but not all graphs are knowledge graphs [13].

It is important to note that both ontologies and data are part of the knowledge graph. Ontologies define the schema and structure, specifying the concepts ans the relationships within it, while data enriches the graph with information from several sources.

There are several knowledge graphs that are widely used, such as DBpedia [14], a knowledge graph extracted from data from Wikipedia and with an RDF format. Another example is the Google Knowledge [15], which presents information in response to search queries on Google, based on the search behavior of web users. Knowledge graphs are used in various domains, including Finance, Healthcare, and Distribution. In finance, they are used for detecting fraud and analyzing the flow of money for their clients [16].

The use of knowledge graphs offers several advantages in different domains. First, they can handle heterogeneous data, which is essential in the context of data science and therefore in this thesis. Second, they support inference and reasoning, allowing the discovery of hidden knowledge and the derivation of new knowledge from existing data. Finally, they provide a structured way to represent knowledge, enabling the integration of data from different sources and domains.

However, knowledge graphs also present challenges and limitations, mainly related to the quality of the data and the construction and maintenance of the graph.

2.4.1 Knowledge Graph and RDF

Knowledge graphs are often represented in RDF format that provides a structured and interoperable way to represent data. This format facilitates the representation of a wide range of knowledge on the Web. IRIs play a crucial role as they provide a unique identifier for resources, enabling the representation of distributed graphs with data from different sources.

With RDF, data are interconnected from diverse sources on the Web, facilitating the discovery of new knowledge via knowledge graphs. RDF also provides a flexible data representation framework that enables the integration of new data and the evolution of the graph using ontologies such as RDF Schema and OWL. A significant advantage of RDF is its ability to support semantic inference through complex queries, enabling the extraction of valuable information from the graph.

Therefore, RDF is a powerful tool to represent knowledge graphs, enabling the integration and interconnection of data from different sources, as well as the discovery of new knowledge through reasoning and inference.

2.5 Data Lineage

The main objective of this thesis is to address data lineage challenges. This underscores the need to introduce the concept of data lineage and its importance in our research.

Let us return to the definition of data lineage mentioned in the introduction, which states that "Lineage, or provenance, in its most general form, describes where data came from, how it was derived, and how it was updated over time" [1]. Data lineage is essential in data science as it provides information about the origin, history, and transformation of data, enabling data scientists to understand the data and make informed decisions. It is used to track the flow of data from its source to its destination, providing insights into the data quality, reliability and reproducibility of the results. As noted above, lineage can also be called provenance, and in the context of this thesis, we also use the term provenance to refer to the origin and history of data.

Provenance information gathers various details, including the author of a dataset, the date of creation, the transformations applied to the data, etc. For instance, knowing such information enables a better understanding of the quality of the data, as well as the semantic implications.

To achieve the main objective of this thesis, we need to represent provenance data in a structured way to enable the extraction of valuable information from the data. This is where the provenance ontology [7] comes into play, as it provides a formal representation of provenance data as RDF. The following section introduces the provenance ontology and its importance in the context of this research.

2.5.1 Provenance Ontology

Provenance, in the context of data, refers to the origin, the history, and transformations applied to data throughout their lifecycle. A provenance ontology is a formal representation of such data.

The provenance ontology known as PROV Ontology (PROV-O) is an OWL2 ontology that enables the mapping of the provenance data model to RDF. This ontology is composed of a set of classes, properties, and restrictions designed to enable the structured representation of provenance data. OWL2 is the second version of the Web Ontology Language, which provides a formal way to define ontologies and vocabularies as explained in Section 2.3.

PROV-O provides a framework of Starting Point classes and properties to represent provenance data, which can be further extended to meet specific requirements. Two levels of extension exist: the Expanded framework, followed by the Qualified framework. The Starting Point framework is illustrated in Figure 2.6.

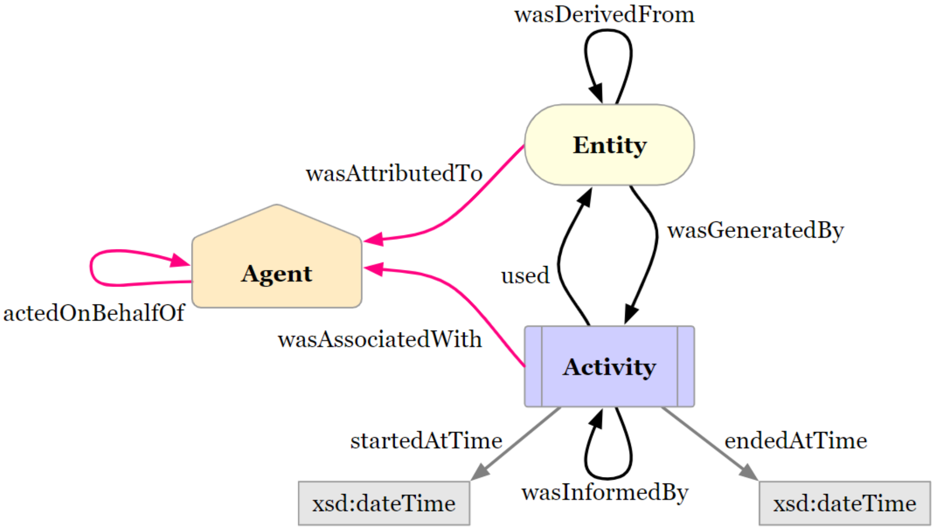


Figure 2.6: PROV-O Starting Point classes and properties [7].

At the Starting Point level, the base ontology includes three fundamental classes:

- `prov:Entity` represents an entity, which is a physical, digital, conceptual, or other kind of thing.
- `prov:Activity` represents an action that occurs over a period of time.
- `prov:Agent` represents someone or something that is attributed as responsible for the occurrence of an entity.

These three classes are interconnected through properties, as shown in Figure 2.6. For example, the `prov:wasGeneratedBy` property links an `Entity` to an `Activity`, indicating that the `Entity` was generated by that specific `Activity` [7].

To clarify, the `prov` prefix is used to refer to the PROV-O namespace. It is a shorthand to avoid repeating the full namespace IRI each time a class or property is mentioned.

2.6 Mapping and Querying Languages

In this section, we introduce two essential elements for the achievement of the research objectives: the query language SPARQL and the mapping language *RDF Mapping Language* (RML). These tools play central roles in materializing our approach.

2.6.1 RML

RML stands for RDF Mapping Language and is the extension of R2RML (RDB to RDF Mapping Language), a W3C Recommendation used to map relational databases to RDF [17]. RML is a language that allows the mapping of heterogeneous data structure, such as CSV, JSON, and XML to RDF data models in a declarative way. Although not a W3C Recommendation, it is used in the context of this thesis to map JSON data to an RDF data model to build a knowledge graph [18]. The RML community has started to consolidate its efforts and recently published a new version of the specification [19] with the aim of standardizing it.

Several tools are available for generating RDF data from JSON using RML. These tools provide functionalities for efficiently mapping heterogeneous data structures, facilitating the creation of knowledge graphs. The following sections present two notable tools for generating RDF data from JSON using RML.

RMLMapper

RMLMapper is a tool which executes rules to generate linked data from heterogenous data sources. It is a Java-based tool which can be used to generate RDF data from JSON data using RML rules [20].

Morph-KGC

Morph-KGC is a tool that constructs RDF knowledge graphs based on heterogenous data sources. This tool supports both RML and R2RML mapping languages and is built on top of a Python library: Pandas.

Morph-KGC offers various features, including the ability to declare transformation functions via RML-FNML [21] and to define RML views to create virtual datasets based on the original data. The latter enables the use of functions, complex joins ans mixed content handling using the SQL query language. This task is possible only over tabular data and JSON files [22].

2.6.2 SPARQL

The query language and protocol used to query RDF data is SPARQL. SPARQL stands for *SPARQL Protocol and RDF Query Language*, which is a recursive acronym. This declarative language allows executing queries over RDF datasets. It encompasses various functionalities, and among them are the following:

- Retrieving data from RDF datasets.
- Updating RDF datasets (Deletion, Insertion, Modification).

- Creating new RDF datasets.

SPARQL enables the use of different types of query, such as SELECT, CONSTRUCT, DESCRIBE, and ASK. The SELECT query retrieves data from a dataset, while the CONSTRUCT query generates new triples to form a new RDF graph based on the query results. The DESCRIBE query provides a description of a resource that matches the pattern defined in the query, and the ASK query checks if a pattern or condition holds true in an RDF graph. SPARQL encompasses both a query language and a protocol, enabling communication and interaction with RDF data sources by applying queries and updates operations.

A SPARQL query is composed of different parts. It starts with the definition of the prefixes used in the query followed by the result clause (e.g., SELECT) specifying the variables to be returned. Datasets to be queried can be defined using the FROM and FROM NAMED clauses. The mandatory WHERE clause is used to define the pattern to be matched in the RDF dataset, and the query can be further modified with solution modifiers such as ORDER BY and LIMIT [8].

An example of a SPARQL query is illustrated in Listing 2.2. In this query, we aim to retrieve the first name, last name, nickname, and birth date of a person from the dataset. In the WHERE clause, the ?person variable is used to represent a person, and the keyword ”a” serves as a shorthand for the rdf:type predicate. The FILTER keyword is used to apply a condition on the birth date: we only want to retrieve the persons born after January 1, 1900. The OPTIONAL keyword signifies that the nickName is optional information. If it is not present in the dataset, the query will still return the first name, last name, and birth date of the person. Finally, the results are ordered by last name and first name.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?firstName ?familyName ?birthDate
WHERE {
    ?person a foaf:Person ;
    ?person foaf:firstName ?firstName ;
    ?person foaf:lastName ?familyName ;
    ?person dbo:birthDate ?birthDate .
    OPTIONAL {
        ?person foaf:nickname ?nickname .
    }

    FILTER (?birthDate >= "1900-01-01"^^xsd:date)
}
ORDER BY ?familyName ?firstName
```

Listing 2.2: An example of a SPARQL query.

SPARQL offers a wide range of functionalities for querying RDF datasets, making it a powerful tool for extracting valuable information. Its abilities extend with features such as subqueries and aggregate functions that enable to perform more complex queries. In addition, SPARQL allows the execution of federated queries, enabling querying of multiple endpoints and integration of external services [8].

In SPARQL, a property path is a way to traverse a graph by following a sequence of edges. The trivial case considers a path of length 1, which is a triple pattern. The property path can be used to represent more complex patterns, to get more information from the graph. For example, we can use SPARQL queries to access the friends of friends of a person, or the siblings of a person. The Listing 2.3 illustrates an example of a SPARQL query using property paths. In this query, we aim to retrieve the names of friends of friends of a person named Lara. The property path is a powerful feature of SPARQL that allows for the representation of complex patterns in a simple way [8].

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?friendName
WHERE {
    ?person foaf:Name <Lara> ;
            foaf:knows/foaf:knows ?friend .
    ?friend foaf:Name ?friendName .
}
```

Listing 2.3: An example of a SPARQL query using property path.

The query in the Listing 2.3 can also be written with explicit triple patterns and variables as illustrated in Listing 2.4.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?friendName
WHERE {
    ?person foaf:Name <Lara> ;

    ?person foaf:knows ?x ;
    ?x foaf:knows ?friend ;

    ?friend foaf:Name ?friendName .
}
```

Listing 2.4: An example of a SPARQL query using property path.

In this thesis, SPARQL is the tool used to query the knowledge graph built from provenance data extracted from Jupyter notebooks. This highlights the importance of SPARQL in the context of this work and its significant role in data science research.

2.7 Summary of Background

This chapter introduced fundamental concepts and tools that are be used in the following chapters. We introduced the notions of graph data models and RDF, highlighting their significant role in the structuring and interconnecting data. In addition, we discussed the importance of ontologies and vocabularies in the context of knowledge graphs and their use in representing knowledge.

Furthermore, we explored the concept of data lineage, which is the main focus of this thesis. Finally, we presented the tools necessary for achieving the research objectives, including RML for mapping heterogeneous data to RDF and SPARQL for querying RDF datasets.

Chapter 3

Related Work

The research question addressed in this thesis highlights several aspects related to the representation of data and how knowledge can be harnessed. This work, situated at the intersection of several research fields, is centered around the use of provenance information and knowledge graphs to tackle challenges related to data lineage in data science. Data science is the most obvious field, as the thesis aims to analyze, interpret, and extract knowledge from Jupyter notebooks. Data management can be mentioned, as data lineage is a key aspect of data management: it is essential to the traceability of data. Knowledge graphs are also considered as a key component of this work, as they have been chosen as the approach to achieve the objectives of the thesis.

The evolution across time in this field is traced to obtain a detailed understanding of recent advancements. In this chapter, we explore previous works that have paved the way for the subsequent research, highlighting the most significant contributions that have shaped our current understanding of the problem.

To tackle the objectives of this thesis, the research question is considered in the broad context of existing literature on the subject. In the subsequent sections, we delve into the most relevant works published in the field of data science with challenges related to data lineage. Additionally, the most significant contributions to knowledge graphs and provenance information are explored.

3.1 Exploring Key Challenges in Data Science

The field of data science has been the subject of numerous studies due to its exponential growth over the past few years. This interdisciplinary domain, which combines statistics, computer science, algorithm development, and domain knowledge, aims to extract knowledge from data. Effective data management plays a central role in the success of data science projects [23, 24].

Numerous challenges in data science have been identified in the literature, potentially negatively impacting the work of data scientists and the quality of their results. In this section, some of the most significant challenges identified in data science are explored. We want to have a better comprehension of the challenges researchers are faced with, by examining some of them closer.

Victor Cuevas-Vicenttín et al. have addressed several challenges in the context of provenance and scientific workflows [6]. They highlighted open issues, including the visualization and analysis of provenance data, that were not largely explored. They also discuss the potential of those techniques to improve the understanding of scientists and help them to debug their tasks. Moreover, the authors identify the need for better integration and standard representation of provenance data to facilitate their sharing.

The challenges of provenance in the context of scientific workflow management system have been discussed by K. Alam et al. [4]. The authors highlight that provenance information can become large and complex in scientific workflows to enable reproducibility, sharing and reuse of knowledge. They also emphasize the challenges of maintaining and storing such data effectively.

Besides challenges related to provenance, others appear in more specific contexts of data science, such as code duplication in Jupyter notebooks. This issue has been addressed by Andreas P. Koenzen et al. [25]. The authors have analyzed multiple Jupyter notebooks to identify code duplication and recurring patterns. They found out that code duplication is a common practice for expediting experimentation processes, despite its effects on code readability. Indeed, the duplication of code increases the burden on the maintenance process.

Through an examination of the various stages of data analysis processes, significant challenges associated with the use of pipelines to execute these processes are identified. In data science, pipelines are widely used to execute data processing workflows. However, these pipelines show limitations in terms of reproducibility, compatibility, and recovery of data analysis processes. Additionally, the abstract composition and reuse of pipeline steps based on a semantic description remains an unresolved challenge. This issue is related to the data lineage that aims to track the data flow across the steps of the pipeline [26].

Hassan Hussein et al. have also mentioned the importance of reproducibility and the challenges it presents in the context of scientific work. They explore the relation between reproducibility and elements in a work such as data, scripts, and simulations. The authors mentioned that there are four main pillars that impact the reproducibility: Availability, accessibility, linkability (using ontologies to link elements from different sources), and license [5].

3.2 Leveraging Ontologies and Knowledge Graphs in Data Science

In data science, the resolution of complex challenges often needs to be addressed through advanced and innovative approaches. Among these, ontologies and knowledge graphs appear to be powerful tools for representing, managing, and leveraging knowledge across various domains, as presented in some articles presented in this section. These approaches offer a semantic structure to extract the richness of relationships between data to facilitate the resolution of complex problems.

Ontologies give a formal representation of knowledge by enabling a common understanding of concepts and their relationships in a domain. Knowledge graphs, on the other hand, are a way to represent knowledge in a graph structure, enabling an intuitive way to explore the knowledge and facilitating the discovery of new insights. These approaches have been used in the literature to address different challenges and are related, as ontologies act as a schema for knowledge graphs. This section explores some of the most significant works that have used ontologies and knowledge graphs in such contexts.

A recent work discussed the use of ontologies in different dimensions, focusing its research on applying ontologies to model data science processes themselves [26]. In their work, the authors have used ontologies (OBOE [27] and OBCS [28]) to annotate, enrich, and document data pipelines, as well as explored approaches to link metadata semantically to the data and transformations. They have used Apache Beam to integrate semantic descriptions in pipelines to facilitate research tasks and reuse of pipelines. However, the authors do not provide information about queries on semantic metadata. Moreover, they do not introduce the aspect of reasoning, which could be a key aspect in certain contexts. Indeed, reasoning allows for harnessing the potential of semantic metadata by facilitating the deduction of new information, for example. Finally, no use cases of their ontology in a real-world scenario are mentioned in their work.

Jupyter notebooks are a popular tool in data science where results might change between each execution. Sheeba Samuel et al. highlight reproducibility issues due to that as mentioned in Section 3.1 [29]. They propose an ontology called ProvBook to describe Jupyter notebooks with provenance information. The ontology the authors have developed,
REPRODUCE-ME, is based on PROV-O [7] and `P-Plan` [30] to describe Jupyter notebooks with provenance information. Their work aims to track the complete path of scientific experiments for trust and reproducibility. For example, they have access to the start and the end of execution of each cell, to changes in the markdown cells, and to the output of each cell. The tool the authors have developed, ProvBook, uses their ontology to let users share their notebooks with provenance information in RDF format. This work brings a new perspective on how to represent Jupyter notebooks with provenance information. However, they

only provide base information about a notebook and do not include details about its content, such as datasets used, author(s) of the notebook, imported libraries, etc. Moreover, they do not provide any information about inference or logical deduction in their work.

The use of knowledge graphs in data science has been explored by several authors. Hassan Hussein et al. have proposed to improve reproducibility by defining a semantic template for knowledge graphs that provides a standard way to describe the elements of a work, including data, software scripts, and simulations [5]. The research aims to ensure the reproducibility of a work through knowledge graphs with the introduction of a semantic template to build a standard way to describe the elements. The authors use the template and a score of reproducibility to evaluate the reproducibility. This score is computed from four pillars: availability, accessibility, linkability, and license. Open Research Knowledge Graph (ORKG)[1] gives access to a template system to define the structure of contributions. The authors have shown that the use of knowledge graphs can improve the reproducibility of a work, but the use and filling of the template can be complex and time-consuming. Moreover, the score computed based on the four pillars does not capture all the aspects of reproducibility, opening up uncertainty related to the reliability of results.

In another context, Shivani Choudhary et al. have used knowledge graphs as embeddings to represent knowledge for several applications, such as predicting missing information in knowledge graphs completion and facilitating question answering [31]. While knowledge graph information is structured, its consumption can be challenging in real-world applications. The authors use embeddings to represent the knowledge of knowledge graphs in a low-dimensional vector space, where the resulting vectors correspond to graph properties. The authors focused on translation-based models, which are an addition of vectors, and showed how this model improved over time. Furthermore, the authors have also described real-world applications of their work with knowledge graphs embeddings, including link prediction and triple classification.

Finally, a recent work by I. Dasoulas et al. has proposed a set of tools composed of an ontology (MLSO), taxonomies (MLST), and knowledge graphs (MLSea-KG) to improve the search, explainability, and reproducibility of ML pipelines [32]. This set is called MLSea and integrates ML experiment data and metadata. The ontology and the taxonomies provide a flexible schema to represent ML pipelines, implementations, etc. The knowledge graph, on the other hand, is used to discover and explore the ML data. To define mapping rules between the ontology and the data, the authors have used RML with the help of Morph-KGC. With these tools, they have shown, for example, the possibility of finding a dataset with its relevant code notebook and the scientific papers related to it.

Certain research discussed above directly relates to the objectives of this thesis and will be

[1] `https://orkg.org/`

used as a basis for the development of our approach in the subsequent chapters. The work of Sheeba Samuel et al. [29] will be mainly used as a reference to develop our approach.

3.3 Summary of Related Work

Challenges related to provenance, code duplication, and data analysis pipelines highlight the complexity of data science and underscore the need to develop approaches to overcome these challenges.

Ontologies and knowledge graphs have been used in various contexts to address different challenges in data science. Based on the research discussed in Section 3.2, we can see that these approaches have been used to represent, manage, and leverage knowledge across different domains. These works have shown the potential of ontologies and knowledge graphs to address complex challenges in data science, such as reproducibility and provenance information in Jupyter notebooks.

Chapter 4

Knowledge Graph Design

The design of the knowledge graph intended to represent the information contained in notebooks establishes the initial stage in leveraging knowledge graphs and provenance data to address challenges related to data lineage. The design of the knowledge graph is crucial to ensuring the correct representation of the information contained in the notebooks. It also influences the querying process for the questions we want to address.

In this chapter, the design and structure of the knowledge graph are described, as well as the strategies employed in its construction. Additionally, different subjects are discussed, such as the data sources used, their formats, and the extracted data.

4.1 Data Sources for the Knowledge Graph

The primary data sources used in this work are notebooks, but additional sources of information can complement the information extracted from the notebooks. Public APIs offer this valuable means to gather more information, thus complementing and enriching the knowledge graph with external information based on the data contained in the notebooks.

4.1.1 Jupyter Notebooks

Jupyter notebooks serve as the primary source of information to be represented and integrated in the knowledge graph. These notebooks contain a rich set of information, including code cells, output cells, markdown cells, and metadata. Additionally, they also contain implicit information, such as the relationships between notebooks, the dependencies between cells, and provenance information.

Notebooks are stored in JSON format, which offers a flexible, hierarchical, and structured way of representing semi-structured data, unlike relational databases that have a tabular structure. However, the structure of the JSON format can vary significantly between notebooks, depending on the environment in which they were created and executed. It means that the keys can be different between two JSON files. For example, consider two cells from two different notebooks from the KGTorrent dataset [33]. The first cell is represented in Listing 4.1, and the second in Listing 4.2. The structures of the JSON format in the two notebooks are different; Notebook 2 includes a unique id in the metadata, while Notebook 1 contains no metadata.

```
...
{"cell type": "code",
"execution_count": 3,
"metadata": {},
"outputs": [],
"source": [
    "def get_df(dir_15_name,
       dir_19_name):\n",
    ...
    ]
}
```

Listing 4.1: A cell in Notebook 1 of the KGTorrent dataset.

```
...
{"metadata":{
    "id":"g1XbinSCkjfv",
    "trusted":true
    },
"cell_type":"code",
"source":
    "NAMES_LIST = \"/kaggle/
       input/a3data/y
    ..."
}
```

Listing 4.2: A cell in Notebook 2of the KGTorrent dataset.

Understanding these various structures is crucial when extracting data from notebooks to construct a knowledge graph. Two structures have been identified in the above examples; nonetheless, other variations exist. The goal of this thesis is to assess the feasibility of representing notebooks as a knowledge graph to formulate queries about provenance. Therefore, in this work, we focus on notebooks created and executed on Google Colab[1] to ensure a consistent structure of the JSON format of the notebooks and facilitate the extraction of information.

The key-value structure of a notebook created and executed on Google Colab is the following:

- **cells**: list of cells in the notebook.
  Each cell is represented by the following elements:
  - **cell_type**: type of the cell (code or Markdown).
  - **execution count**: position of the cell in the execution order (null if the cell is not executed).
  - **metadata**: metadata of the notebook, including the cell id and the output id

[1] https://colab.research.google.com/

when there is an output.

  - **source**: content of the cell.

- **metadata**: metadata of the notebook, including the kernel information, language information, and metadata about Google Colab. All the previous cited information do not always exist in the metadata.
- **nbformat**: version of the notebook format.
- **nbformat_minor**: minor version of the notebook format.

A typical structure of a Jupyter notebook in JSON format is shown in Listing A.1 in the Appendix.

4.1.2 Public APIs

Public APIs offer a valuable source of information to enrich a knowledge graph with external data derived from the content of notebooks. In this work, we use three different APIs to enrich the graph representations of notebooks with additional information about libraries, datasets, and authors within the graphs.

The first API is the PyPi API [34], which retrieves information about the libraries used in the code cells of notebooks. Based on the name of the library, this API provides details such as the required Python version of a library, the author of a library, a homepage where the library is presented, etc.

Second API is the GitHub REST API endpoints for users [35]. This API is used to retrieve additional information about the author of a notebook. Based on the username of a GitHub account, the API returns the URL of the GitHub account, an associated email address, information about repositories, and more.

The third API is the Kaggle API [36], which provides access to details about datasets that are manipulated in notebooks. These details includes the title of the dataset, the license of the dataset, the author of the dataset, the number of downloads, and a description, based on the reference of the dataset.

Some information from these APIs has been selected to be extracted and represented in the knowledge graphs. This information is collected through an HTTP GET request to the APIs. The response is in JSON format, which is parsed to extract the desired information. In the PyPi API, the information that is extracted is the homepage and the required Python version of the library. In the GitHub API, the information is the GitHub profile URL of a user. And in the Kaggle API, the information that is extracted is the title and the license of a dataset.

The integration of data collected from public APIs enriches considerably the richness of available data in the knowledge graphs. Thanks to these external sources, all the criteria mentioned in Section 2.4 are met, and the graph is thus a knowledge graph. By incorporating this external data, we create a more complete and dynamic representation of the information, offering new opportunities for analysis and exploration of the data contained in the notebooks.

4.2 Structure of the Knowledge Graph

To represent the information extracted from notebooks in the knowledge graph effectively, a coherent and clear structure is necessary. As explained in Section 2.3, the information can be structured using an ontology that defines the classes and properties of the information to represent. The structure of the knowledge graph used to represent the provenance information is designed to rely on the PROV-O ontology introduced in Section 2.5.1.

The design of the structure must consider the conventions defined by the PROV-O ontology. To be compatible with this ontology, the data must be organized in a manner that clearly identifies each element (cell, dataset, author, etc.) as either an `Entity`, an `Activity`, or an `Agent`. The data structure must assign each element to one of these classes and describe their relationships and interactions to ensure their representation in a semantically coherent way as part of the PROV-O ontology.

Before diving into the representation of a notebook with the PROV-O ontology, it is essential to provide more explanation about PROV-O itself. This ontology is composed of three different levels, with the one we base our work on being the *Starting Point*, serving as the basis of the ontology. However, for a complete representation of the provenance information, the second level has been considered. This level extends the base ontology by introducing additional classes and properties. The second level, called the *Expanded Terms*, is represented in Figure 4.1. In the *Expanded Terms* derived from the base ontology, we consider the class `Collection` that is a subclass of `Entity`, and represents a collection of entities. Another relevant class is `Person`, a subclass of `Agent`, representing an individual. These two classes, along with their associated properties represented in the second level of the ontology, contribute to defining the desired structure.

The structure of the knowledge graph can be defined on the grounds of the PROV-O ontology by considering the *Starting Point* and the few terms introduced above from the *Expanded Terms*. We begin by considering each notebook as a `Collection` of cells where each cell is an `Entity`. Thus, every cell is linked to the notebook to which it belongs. Additionally, each notebook is associated with its author(s), which is represented as `Person`. It is also related to all the libraries mentioned in the source code of each cell of the notebook that are represented as `Agent`. In a notebook, it is possible to access manipulated datasets, each represented as

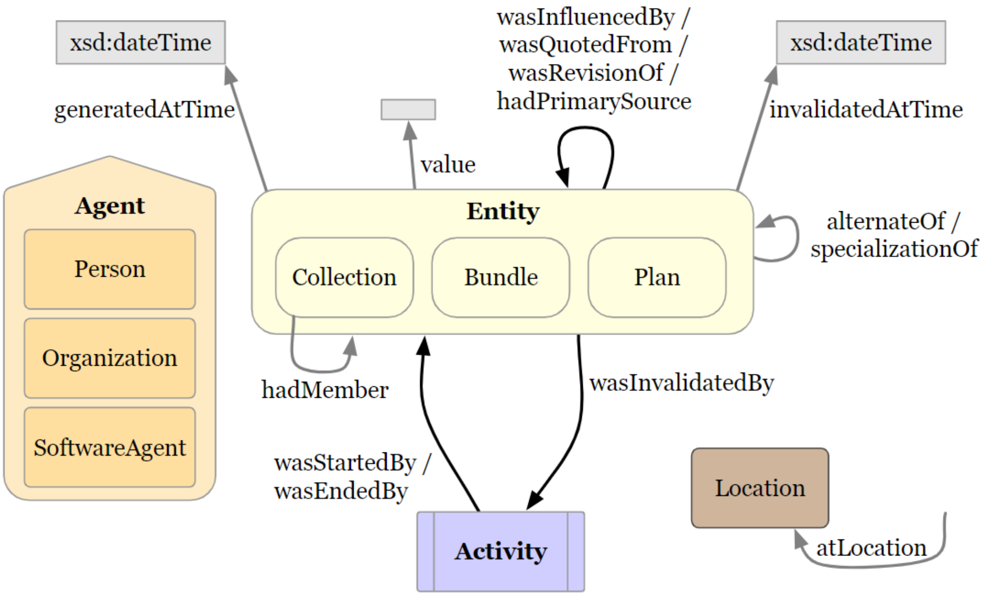


Figure 4.1: The Expanded Terms PROV-O Ontology [7].

Entity. Furthermore, a notebook can also be associated with licenses that are also represented as Entity. These licenses can either be extracted from the code as a comment, as explained in Section 4.3, or obtained from datasets manipulated in the code. The execution of a cell is an Activity that is associated with the cell that has been executed. Finally, the output of a cell is considered as an Entity and is related to the cell that has produced it and to the execution of the code that has generated it. The schema of the different elements of a notebook, sorted by categories, and their relationships is shown in Figure 4.2.

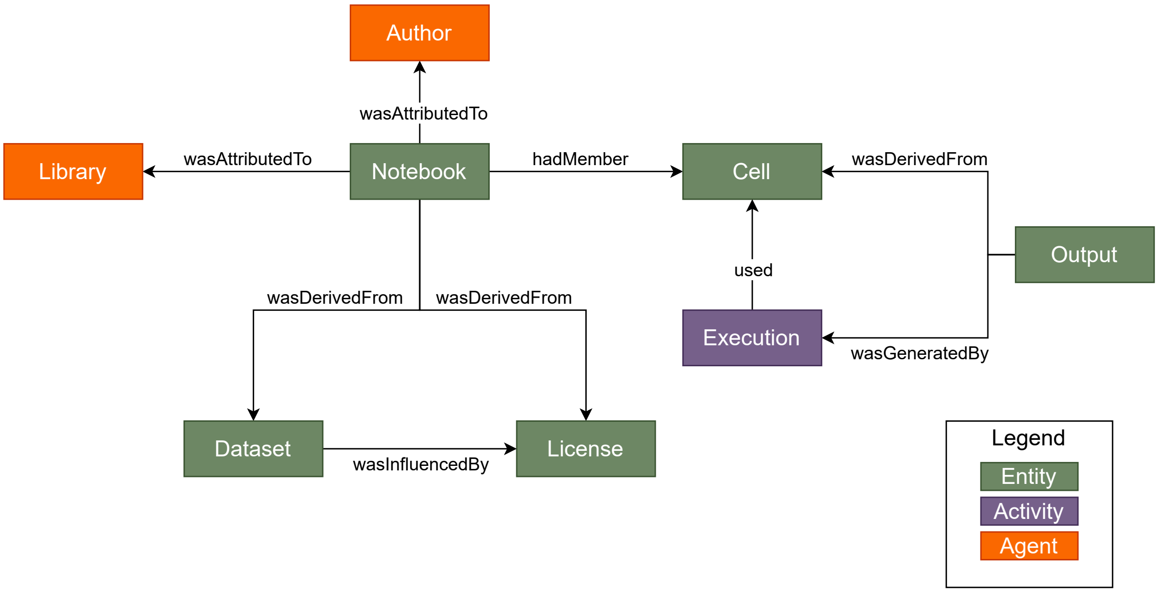


Figure 4.2: Structure of the knowledge graph.

All other collected information is associated with the three main classes mentioned above but is stored as literals in the knowledge graph. For example, the date of the notebook is represented as a literal related to the notebook using the property `generatedAtTime`, as specified in the expanded terms of the PROV-O ontology.

4.2.1 Extension of the PROV-O Ontology

The PROV-O ontology is a generic ontology that is used to represent provenance information. However, all the information that can be extracted from a notebook is not representable solely with the PROV-O ontology. As this ontology is meant to be extended for specific application domains, we have decided to undertake its extension to represent all the desired information.

First, the FOAF (Friend of a Friend) vocabulary is used to represent the information about the authors of the notebooks. As explained, the authors are represented as `Person` in the PROV-O ontology. The use of the FOAF vocabulary facilitates the representation of information about the authors [12].

Second, some of the information that can be extracted is specific and not representable with any known ontology, such as the type of cell (code or Markdown). Therefore, we have decided to create our expanded terms. These terms are represented in Table 4.1.

To ensure the coherence and interoperability of our representation of knowledge, we have introduce a `namespace`, which is used to identify newly created expanded terms in the knowledge graph. This one is the following: `http://example.com/knowledge-graph/`. The prefix associated with the namespace in this work is `myns`.

4.3 Data Enriching with Source Code Annotations

For each key-value pair in the JSON format of a notebook, there is relevant information to extract and represent in the knowledge graph. Nevertheless, our interest also extends beyond these key-value pairs to include the content within each cell, more specifically the source code, which serves as the core of the notebook, retrieved from the `source` key of each cell. However, the extraction of information from the source code, such as libraries, authors, or datasets, is not well-structured and may even be absent. To address this challenge, we have established an imposed structure to facilitate the process.

The concept of the imposed structure is to define a specific format for the information that need to be extracted from the source code. This information is written as a key-value pair in comments:

```
# Key :  Value
```

Table 4.1: The predicates created to extend the PROV-O ontology. The `hasExecutionCount` predicate has a `string` for range because it can have a null value when the cell is not executed.

Predicates	Definition	Domain	Range
`myns:hasCellType`	The type of the cell (code or markdown)	`Entity`	`string`
`myns:hasExecutionCount`	The execution position of an activity in a sequence of activities	`Entity`	`string`
`myns:hasOutputType`	The type of the output (stream or display)	`Entity`	`string`
`myns:hasLanguage`	The programming language used in the notebook	`Entity`	`string`
`myns:hasLanguageExtension`	The extension file of the language	`Entity`	`string`
`myns:hasLanguageVersion`	The version of the programming language used in the notebook	`Entity`	`string`
`myns:hasDefinedFunction`	The name of the function defined in the code	`Entity`	`string`
`myns:hasCalledFunction`	The name of the function called in the code	`Entity`	`string`
`myns:hasGithubAccount`	The GitHub account of the user who created the notebook	`Agent`	`string`
`myns:hasGithubURL`	The github URL of the Person	`Person`	`string`
`myns:hasIRI`	The IRI of the entity	`Entity, Agent`	`string`
`myns:hasRequiredVersion`	The required version of the language for a library	`Agent`	`string`
`myns:hasSourceCode`	The source code of the cell	`Entity`	`string`

where `Key` is the keyword that identifies the information, such as `Date`, `License`, `Dataset`, and `Value` is the information that needs to be extracted.

Consider an example of a code cell represented by Listing 4.3. This cell contains information about the date, the authors, and the URI of the notebook on Google Colab. The keys are identified by the keywords `Date`, `Author`, `URI`. The values can be extracted and processed if needed before being represented in the knowledge graph.

```
    "source": [
      "# Date : 12/01/2024\n",
      "# Author : Alixia Birtles - GitHub : alixiabirtles\n",
      "# Author : Lara Birtles\n",
      "# URI : https://colab.research.google.com/drive/19
```

```
         Ce_h7_oxxphYomDEksc8N04vd-3RL9p\n",
      "\n"
    ]
```

Listing 4.3: Representation of the structure of the information in the source code of a cell.

This way of collecting information provides additional information about the notebook, complementing the base data extracted from the JSON format structure.

4.4 Overview of the Extracted Data

Considering the basic structure of a notebook, with the source code and the additional APIs, notebooks provide a rich source of information. This richness of data can be extracted and incorporated in the knowledge graph.

Consider first the information that we are interested in, which primarily includes data related to the structure of the notebook, such as the unique identifier of each cell, the execution order, the output, and information about the notebook itself. These data enable us to understand the structure of the notebook and the dependencies between the cells.

Then, the source code of the cells provides further details about the content of a notebook and tells a story about what the notebook is about. It contains information about used libraries, manipulated datasets, notebook authors, and datasets licenses, etc.

Finally, more detailed information based on those collected in a notebook can be added to the knowledge graph. This information is extracted from public APIs as specified in Section 4.1.2.

4.5 Summary of Knowledge Graph Design

This chapter presented the data sources considered in this work, which include Jupyter notebooks created and executed on Google Colab. As we have shown that notebook formats vary depending on the environment in which they are executed, we have focused on notebooks from Google Colab. Moreover, three public APIs are used to enrich the information extracted from the notebooks: the PyPi API, the GitHub REST API endpoints for users, and the Kaggle API. The combination of all these data sources ensures that the knowledge graph is a true one, as one of the criteria to consider a graph as a knowledge graph is to aggregate information from multiple data sources.

The information extracted from the notebooks and the APIs is then structured within knowledge graphs using the PROV-O ontology. Although the base ontology provides foundational structure, we extend it by incorporating additional terms from the second level of

PROV-O in our namespace. This augmentation allows for a more detailed representation of the extracted information. We have also extended the PROV-O ontology with the FOAF vocabulary and with defined terms that are too specific to be represented with the PROV-O ontology.

Finally, an explanation of how the data are extracted from the source code based on an imposed structure is provided. Indeed, the metadata fields in notebooks are often limited and may vary from one notebook to another. Therefore, to consider more data, we have defined a structure to extract information from the source code of the cells. This structure is based on key-value pairs written in comments in the source code. In the discussions chapter, we explain why a uniform format and additional metadata fields might be beneficial to improve interoperability.

Based on all the sources of information and methods of extraction, we have a complete overview of the knowledge graph design and the data that is extracted from the notebooks.

Chapter 5

Implementation

In the context of this work, we have established a structured schema aimed at representing the information collected from several data sources, in particular Jupyter notebooks and public APIs. This schema, referred to as an ontology, is composed of several classes and properties to structure the information in an organized way. This stage of conception, detailed in the previous chapter, is essential to defining the basis of this work and to guiding the implementation of the knowledge graph.

The process of implementation of the knowledge graph starts by collecting the information from the data sources mentioned, followed by mapping this information to the ontology with the help of mapping rules. The final result is a turtle file with RDF mappings. In this chapter, we look into the details of this implementation, highlighting the different stages needed to manipulate the data and represent it in the form of a graph.

The subsequent sections start by contextualizing the importance of this stage of implementation, underscoring its crucial role in the research process. Then the method to lead this implementation is detailed, highlighting the technological choices and the obstacles faced during this process. Finally, the chapter is organized in a manner that provides a complete overview of the implementation process, approaching key aspects, such as the mapping rules of the information to the ontology.

5.1 Overview of the Implementation

The implementation of the knowledge graph is a crucial stage in the research process, as it allows materializing the theoretical concepts defined in the previous chapters. This stage is essential to demonstrating the feasibility of the approach and evaluating the performance of the knowledge graph in representing the information collected from the data sources. The implementation of the knowledge graph is also essential to identifying the limitations of the proposed approach. If the knowledge graph cannot be represented as desired, it will not be possible to use this approach to face the challenges introduced by the research question.

To implement a knowledge graph, several elements must be considered. We need first data sources that have a structure that can be mapped to the ontology. This structure can be tabular, such as in CSV files, or more complex, in JSON files. Then, we need to define the mapping rules between the information contained in the data sources and the classes and predicates defined in the ontology. These rules are defined through a language that allows representing the relations between the different classes, or between a class and a literal. Each mapping rule specifies how the information corresponds to a triplet RDF (subject, predicate, object), also referred to as a mapping. Finally, the data is transformed into a structure that is compatible with the ontology. Additionally, the RDF mappings that represent the relationships between the data are generated in a turtle file that represents the knowledge graph. An example of an RDF triplet representing the information contained in a Jupyter notebook is shown in Listing 5.1.

```
@prefix myns: <http://example.com/knowledge-graph/> .
@prefix prov: <http://www.w3.org/ns/prov#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

myns:Cell_Dfq-kZ57PgaT a prov:Entity ;
        myns:hasCellType "code"^^xsd:string ;
```

Listing 5.1: Example of a RDF triplet representing the mapping of the information contained in a Jupyter notebook. It represents a `Cell` that is of type `Entity` with a `hasCellType` predicate.

5.2 Mapping Rules

The mapping of the information to the ontology is a main step in the implementation of the knowledge graph. This process consists of associating the information collected from the data sources to the classes and predicates defined in the ontology and its extension as detailed in Section 4.2.

The mapping rules are defined with a language that allows for the representation of information in a structured way and defines the relations between the different classes. The language can either be declarative or imperative, depending on the tool used. A declarative language allows one to define the mapping rules in a declarative way, while an imperative language must specify all the operations to perform on the data to transform it into the desired format.

The declarative language is more suitable for the mapping of the information to the ontology, as it allows for defining the mapping rules in a concise and intuitive way. In addition, the mapping rules are detailed independently of the implementation details, which isolates the logic of the mapping from the logic of the data manipulations. Finally, declarative languages are designed to effectively manage the complexity of the mapping process and to improve the reusability of the mappings [37].

In this work, we aimed to use a declarative language to map the information to the ontology. The first choice of language is based on the tools available and the simplicity of the language used to define the mappings. Therefore, RML is considered as it is a standard language for mapping information to RDF, and it is supported by several tools that facilitate the mapping process. In addition, this language allows for defining the mappings for data sources that are not tabular, such as Jupyter notebooks (JSON).

There are several tools that support the RML language, such as RMLMapper and Morph-KGC, which allow executing the mappings and generating the RDF graph from the information sources. In the subsequent sections, the different tools used to map the information with RML are detailed.

5.2.1 RMLMapper

The first tool used to map the information to the ontology is RMLMapper. This tool allows executing the mappings defined in the RML language and generating the RDF graph from the information sources, as explained in Section 2.6.1.

However, the use of RMLMapper has shown some limitations in the mapping process, which has led to the exploration of another tool, Morph-KGC.

5.2.2 Morph-KGC

Morph-KGC is another tool that supports the RML language and allows the execution of mapping rules defined in RML. This tool is more advanced than RMLMapper, as it allows performing operations on the data during the mapping process through the use of the extensions `fno` and `fnml`.

The two extensions aim to use functions to manipulate the data during the mapping process. The `fno` extension is a way to describe and declare functions for use in RML mapping rules. These functions are called to transform the data in a specific way by imposing all the letters to be in uppercase, for example [38]. To connect these functions with the mapping rules, the `fnml` extension is used. Its purpose is to define structure, input parameters, and return types of functions.

In addition, Python `user-defined functions` are also supported by Morph-KGC. The decorators of these functions must be well-specified to link to the parameters from the Python script function to the `fnml` parameters [21, 22].

Consider the piece of code written in RML in Listing 5.2. This code uses a notebook as a data source. It starts by defining a `Logical source` that specifies the data source and the iterator, which is, in this case, the cells of the notebook. The `CellEntity` is then defined, representing the cells of the notebook. It has a `subject map` that defines the subject with a unique identifier created based on the `id` of the cell. Finally, a `predicate object map` is defined to represent the relationship between the cell and its timestamp. The timestamp is not directly available; it is extracted through the call of a function `getTimestamp`, defined in Python. The function `getTimestamp` is linked to the RML mapping rules through the `fnml` extension.

```
@prefix rml: <http://w3id.org/rml/> .
@prefix ex: <http://example.com/> .
@prefix grel: <http://users.ugent.be/~bjdmeest/function/grel.ttl#> .

<CellsSource>
    a rml:LogicalSource;
    rml:source "ExampleJSON.json";              # The data source
    rml:referenceFormulation ql:JSONPath;       # The source format
    rml:iterator "$.cells[*]".                  # Iterate over cells

<CellEntity>
    rml:logicalSource <CellsSource>;            # Source of the data

    rml:subjectMap [                            # Define the subject
        rml:template "http://example.com/code-redundancy/Cell-{metadata.id}";
        rml:class prov:Entity;
    ];

    rml:predicateObjectMap [                    # Define a predicate
        rml:predicate prov:generatedAtTime;
        rml:objectMap [                         # Define the object
            # Apply a function to get the timestamp
            rml:functionExecution <#getTimestamp> ;
        ];
    ];
.

<#getTimestamp>
    rml:function ex:getTimestamp ;              # The py function
    rml:input [
        rml:parameter grel:valueParam ;
        rml:inputValueMap [
            rml:reference "source";             # The input value
        ];
    ] .
```

Listing 5.2: Example of RML mapping rules with Morph-KGC to extract the timestamp from a JSON file using the extensions `fno` and `fnml`.

Therefore, with these extensions, it is possible to extract the information that is located within the source code of the Jupyter notebooks, and map them to the ontology through RML mapping rules.

5.3 Addressing Mapping Tool Limitations with RDFLib

In the previous section, two tools to map the data to the ontology were introduced: RMLMapper and Morph-KGC. However, these tools have shown some limitations in the mapping process while working with Jupyter notebooks.

The limitations related to RMLMapper are described in Section 5.3.1 and is the reason for the introduction of Morph-KGC. However, Morph-KGC also showed some limitations in the mapping process that led to the use of another tool: a imperative language that allows representing the structure of the data as desired. The limitations and the solution are detailed in the following subsections.

5.3.1 Issues with Function Ontologies in RMLMapper

RMLMapper is limited due to its FnO (Function Ontology) implementation, which aims to define functions to manipulate the data during the mapping process [38].

As explained in Section 4.3, we want to have access to the source code in the Jupyter notebooks to extract the information contained in the cells. When we have access to the information that corresponds to the value of the keyword `source` in the JSON file, this needs to be manipulated to extract the relevant data. This operation is complex to perform with RMLMapper, as it is unable to perform operations on the data in the mapping rules, such as deletion of parts of the source code.

5.3.2 File Format Issues in Morph-KGC

Morph-KGC was introduced to address the limitations of RMLMapper. However, an issue occurred with Morph-KGC, which was not encountered with RMLMapper. This issue is linked to the data source. It is known that Jupyter notebooks are stored in JSON format, but have the extension `.ipynb`. When trying to use the JSONPath, like in the line 8 of Listing 5.2, to extract the information from the notebook, Morph-KGC does not recognize the extension and throws an error. This issue does not occur with RMLMapper, as it can directly extract the information from the `.ipynb` file. The problem has been reported to the Morph-KGC team, and they recommended transforming each notebook file to a JSON file before using it as a data source.

This limitation was overcome by converting the notebook files to JSON files before using them as data sources. One could think that this solves the problem, but the goal of a mapping is to ensure that the whole process is self-contained. Renaming files and changing file extensions before transformation do not comply with that principle. That being said, there is another, more important, limitation in the mapping process that led to the use of an imperative language.

5.3.3 Iterator Challenges in Mapping Rules with Morph-KGC

The conceptual idea of the structure was to represent a notebook as a collection of multiple cells as represented in Figure 4.2. As the notebook does not have a unique identifier within the keys of the JSON file, we thus need to create one to represent the notebook. Therefore, we imposed a unique identifier on each notebook, based on the URI of the Google Colab the notebook was executed on. For example, if the URI is `https://colab.research.google.com/drive/19Ce_h7_oxxphYomDEksc8N04vd-3RL9p`, then the last part after the last "/" is kept as the identifier, resulting in: `19Ce_h7_oxxphYomDEksc8N04vd-3RL9p`. This was done by adding a new key-value pair as a comment in a cell of the notebook, as specified in Section 4.3, and extracting it with a user-defined function during the mapping process.

The mapping rules to extract the notebook identifier are shown in Listing 5.3. As a user-defined function is used to extract the notebook identifier, the subject map cannot be defined within a template: a URI based on the identifier. Instead, the subject map of the notebook must be defined as a blank node.

```
<NotebookEntity>
    rml:logicalSource <MainSource>;     # iterator on notebook

    rml:subjectMap [
        rml:class prov:Collection;
        rml:functionExecution <#getNotebookId>; # get notebook id
        rml:termType rml:BlankNode;             # blank node definition
    ];
.

<#getNotebookId>
    rml:function ex:getNotebookId;
    rml:input [
        rml:parameter grel:valueParam;
        rml:inputValueMap [
            rml:reference "source";
        ];
    ].
```

Listing 5.3: Example of RML mapping rules with Morph-KGC to define a notebook.

The issue is related to the missing link there is between the cells and the notebook. Only one cell contains the identifier of the notebook and is related to the notebook, while the other cells do not. When the function `getNotebookId` is called, it returns the identifier of the notebook, but it also state sthat the notebook is only *linked* to the cell where the identifier is written. Even if the data source was initially the whole JSON file, by using the function `getNotebookId`, the data source of the subject, after defining it, is limited to the cell containing the identifier of the notebook. Thus, there are no direct relations between the cells, except the one that contains the identifier of the notebook, and the notebook itself.

However, to respect the structure defined in the Chapter 4, we need to connect all the cells to the notebook. RML does not allow the creation of a predicate linking each cell with the notebook if there is no direct relationship between them. Indeed, RML only supports `equi-joins`, a conditional join based on an equality condition. It does not allow creating a relationship with the other cells of the notebook, as the only equality that exists between the notebook and the cells is the cell that contains the identifier of the notebook. To create the missing links between the other cells and the notebook, `theta-joins` are required, which are not supported by RML. This would let us create a condition that is not based on equality, to link all the cells to the notebook. More detailed explanations of the types of joins mentioned are available in Appendix B. The problem is illustrated in Figure 5.1, where each cell contains its own unique identifier. In contrast, the notebook has access to both its own identifier and the identifier of the cell it belongs to. More specifically, in the figure, the notebook has its identifier that is written in the *cell i* in the notebook; the cell is thus linked to the notebook. Therefore, the notebook has access to its identifier and the identifier of the *cell i.*

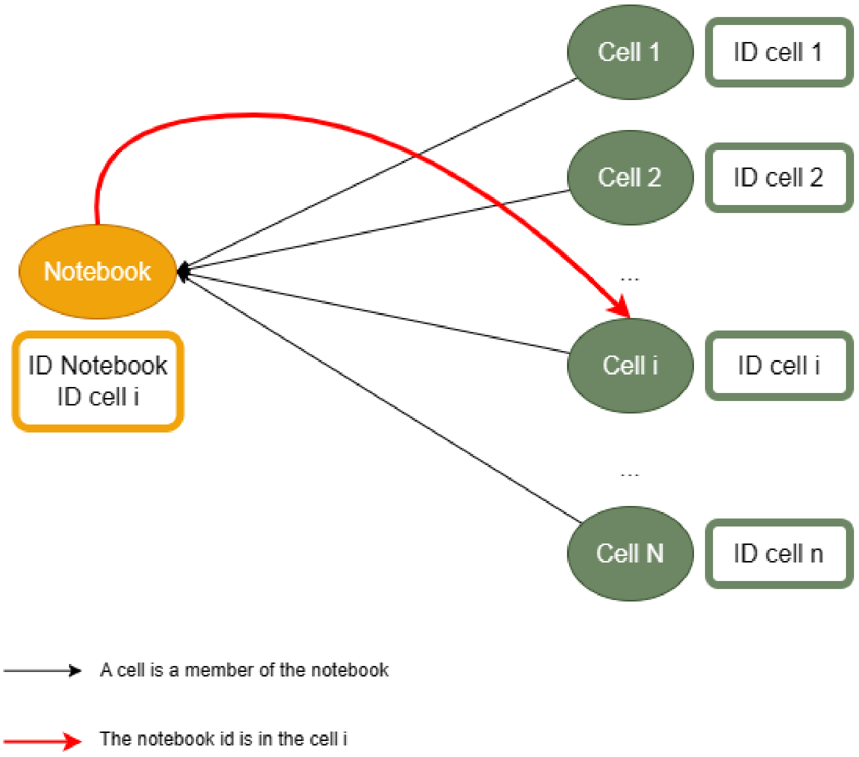


Figure 5.1: Representation of the missing relationship between cells and the notebook to which they belong. The framed identifiers correspond to the unique identifiers accessible from the element the framed identifiers are linked to; either a notebook or a cell.

As the issue is not solvable with RML because the language itself does not support it, we need to use another tool to perform the mappings. The final tool used is an imperative language that allows structuring the data as desired, creating the missing links between the cells and the notebook. This tool is detailed in the following section.

5.3.4 RDFLib to Address RML Limitations

The tool that was finally chosen is RDFlib as it allows structuring the data as desired, thus overcoming the limitations of RML. This tool is a Python package that enable to work with RDF. It is simple to use when knowing Python, and allows for creating RDF graphs and serializing them in different formats such as, Turtle, N-Triples, or RDF/XML [39].

5.4 Construction of the Knowledge Graph

The construction of the knowledge graph is the final step of the implementation. The idea is to create one graph and include in it several notebooks. Therefore, following the RDFLib documentation, an empty graph is first created. Then each `.ipynb` file is added to the graph through several functions that aim at extracting the information from the JSON format and mapping it to the ontology. At each step, the graph is serialized in Turtle format to visualize and check if the information is correctly represented.

5.4.1 RDFLib to Overcome the Limitations of RML

Section 5.3.3 highlighted a limitation faced while using RML to map the information to the ontology. The issue was related to the missing links between the cells and the notebook. This issue can be overcome by using the RDFLib library, as it allows creating instructions to generate the knowledge graph, which includes keeping a reference to each cell with its metadata in a notebook. Therefore, the missing links between the cells and the notebook can be created. The code in Listing 5.4 shows how to create the missing links between the cells and the notebook using RDFLib.

```
for cell in CellsSource:
    # Get the id of the notebook
    id_notebook = getNotebookId(cell.value['source'])

    if id_notebook is not None:
        label_notebook = id_notebook
        notebook_uri = URIRef(f"http://example.com/knowledge-graph/
                                Notebook-{id_notebook}")
    # Get the date at which the notebook has been created
    if getTimestamp(cell.value['source']) is not None:
        date_notebook = getTimestamp(cell.value['source'])

for cell in CellsSource:

        notebook = f"""
        @prefix prov: <http://www.w3.org/ns/prov#> .
        @prefix myns: <http://example.com/knowledge-graph/> .
        @prefix foaf: <http://xmlns.com/foaf/0.1/> .
        @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

        <{notebook_uri}>
        a prov:Collection;
```

```
            prov:label "{label_notebook}"^^xsd:string ;
            prov:hadMember myns:Cell_{current_cell} ;
        .
        """
        graph.parse(data=notebook, format="turtle")

        if date_notebook is not None:
            graph.add((notebook_uri, prov.generatedAtTime,
                        Literal(date_notebook, datatype=XSD.dateTime)))
```

Listing 5.4: Example of Python code using RDFLib to create the missing links between the cells and the notebook.

In this piece of code, we first iterate over each cell to extract the unique identifier of the notebook and the timestamp of the notebook, if there is one. Then, we iterate over each cell again, and we create a notebook for each cell. As the notebook is the same for all the cells, each cell is linked to the same notebook. This is due to the functioning of RDFLib, which does not allow creating multiple instances of the same information. If it is created multiple times, it will only keep one instance of it.

Despite its approach to the mapping process, RDFLib has the capacity to handle this issue effectively. Indeed, the resulting mappings represented in Listing 5.5 align with the desired representation of a notebook and its cells: all the cells belonging to the notebook are linked to it. In this turtle file, the notebook is represented as a `prov:Collection`, and the cells are linked to it through the `prov:hadMember` predicate. Other information, such as the language of the notebook, the language version, and the language extension, is also included in the graph.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix myns: <http://example.com/knowledge-graph/> .
@prefix prov: <http://www.w3.org/ns/prov#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

myns:Notebook_16Jql-jK1yN7SAVAsy7mZtLBfn39D6gCB a prov:Collection ;
    myns:hasCalledFunction "abcd"^^xsd:string,
        "backward"^^xsd:string,
        "item"^^xsd:string,
        "print"^^xsd:string,
        "range"^^xsd:string,
        "torch.linspace"^^xsd:string,
        "torch.no_grad"^^xsd:string,
        "torch.randn"^^xsd:string,
        "torch.set_default_device"^^xsd:string,
        "torch.sin"^^xsd:string ;
    myns:hasDefinedFunction "abcd"^^xsd:string,
        "get_loss"^^xsd:string,
        "get_model"^^xsd:string,
        "get_predictions"^^xsd:string ;
    myns:hasLanguage "python"^^xsd:string ;
    myns:hasLanguageExtension ".py"^^xsd:string ;
    myns:hasLanguageVersion "3.12.0" ;
    prov:generatedAtTime "2024-03-04T00:00:00"^^xsd:dateTime ;
    prov:hadMember myns:Cell_7xnK-vgy52Mo,
        myns:Cell_A7xY7DuX113B,
        myns:Cell_LSMxxEuDH2yt ;
    prov:label "16Jql-jK1yN7SAVAsy7mZtLBfn39D6gCB"^^xsd:string ;
    prov:wasAttributedTo [ a prov:Person ;
            foaf:name "Felix_Mendelssohn" ],
        [ a prov:Person ;
            foaf:name "Lara_Birtles" ],
        myns:Library_math,
        myns:Library_torch .
```

Listing 5.5: Example of RDF graph created with RDFLib to represent the missing links between the cells and the notebook. The cells linked to the notebooks are listed after the `prov:hadMember` predicate. The blank nodes describing a `Person` are represented between brackets [...].

5.4.2 Author as a Blank Node

In Section 2.2, blank nodes were introduced as a way to represent classes that do not have a unique identifier. In the creation of the knowledge graph, `Author` represented as a `Person` in the ontology, was defined as a blank node. This is attributed to the manner in which the author of the notebook is extracted directly from the JSON file. While the name and the family name of the author are accessible, no unique identifier is provided. Blank nodes allow us to create a unique identifier based on this information within the graph without relying on an external identifier. Therefore, if the author of notebook A is the same as the author of notebook B, the blank node will be the same for both notebooks. The code in Listing 5.5 shows how the mappings of blank nodes are defined.

5.5 Summary of the Implementation

This chapter has detailed the implementation of the knowledge graph, highlighting the different stages needed to manipulate the data and represent it in the form of a graph.

The mapping of the information to the ontology was detailed, as were the different tools used to perform the mappings. We faced some limitations concerning the mapping process while trying to create mapping rules with RML. Two tools, RMLMapper and Morph-KGC, that allow writing mappings rules in a declarative language, were tested before switching to an imperative language, as they do not allow designing the graph as desired. RMLMapper was not able to perform operations on the data during the mapping process, while Morph-KGC had iterator challenges and file format issues. The solution to overcome these limitations is the use of RDFLib, a Python package that allows structuring the data as planned in the previous chapter.

Finally, the construction of the knowledge graph was mentioned, detailing some key aspects such as the creation of the missing links between the cells and the notebook, and the representation of the author as a blank node.

Chapter 6

Evaluation and Experimentation

An ontology was defined in the previous chapter as well as mapping rules with the aim of generating the RDF triples to construct the knowledge graph. By considering a dataset with notebooks, a knowledge graph can be generated, evaluated, and queried to answer several questions of interest related to data lineage. In this chapter, the evaluation of a generated knowledge graph is presented, and the experimentation with it in real-world conditions is discussed.

The evaluation phase is essential to ensure the information extracted from notebooks is correctly represented within the knowledge graph and to identify any errors in the mapping process. To conduct this evaluation, a knowledge graph based on two test notebooks is built and visualized using a tool called GraphDB. Several queries are executed on that graph, and their results are analyzed to validate it.

The experimentation phase aims to test the approach under real-world conditions with a dataset composed of notebooks from various sources. The KGTorrent dataset, on which simple queries are executed, is the pillar of this stage. It is a dataset of Jupyter Notebooks in Python from the Kaggle community. Additionally, performance measurements of the generation of the knowledge graph and of the querying process are realized.

6.1 Visualization of the Knowledge Graph

The visualization of the knowledge graph is useful to check if the information is correctly represented and to identify errors in the mapping process. The visualization has been done with GraphDB. It is a graph database that allows visualizing RDF graphs and querying them with the support of RDF and SPARQL [40].

In the knowledge graph, notebooks are of the type `Collection`, and many properties are associated with them, such as the cells they have as members, the libraries they are attributed to, the licenses they derived from, and the datasets they derived from. Figure 6.1 shows a visualization of one of the notebooks that are stored in the graph and its properties. The literal values are not displayed in the directed graph in Figure 6.1, but they can be accessed within the application as a list, by clicking on the node. Another aspect of GraphDB is that it does not allow displaying the blank nodes, which are the authors of a notebook in this case.

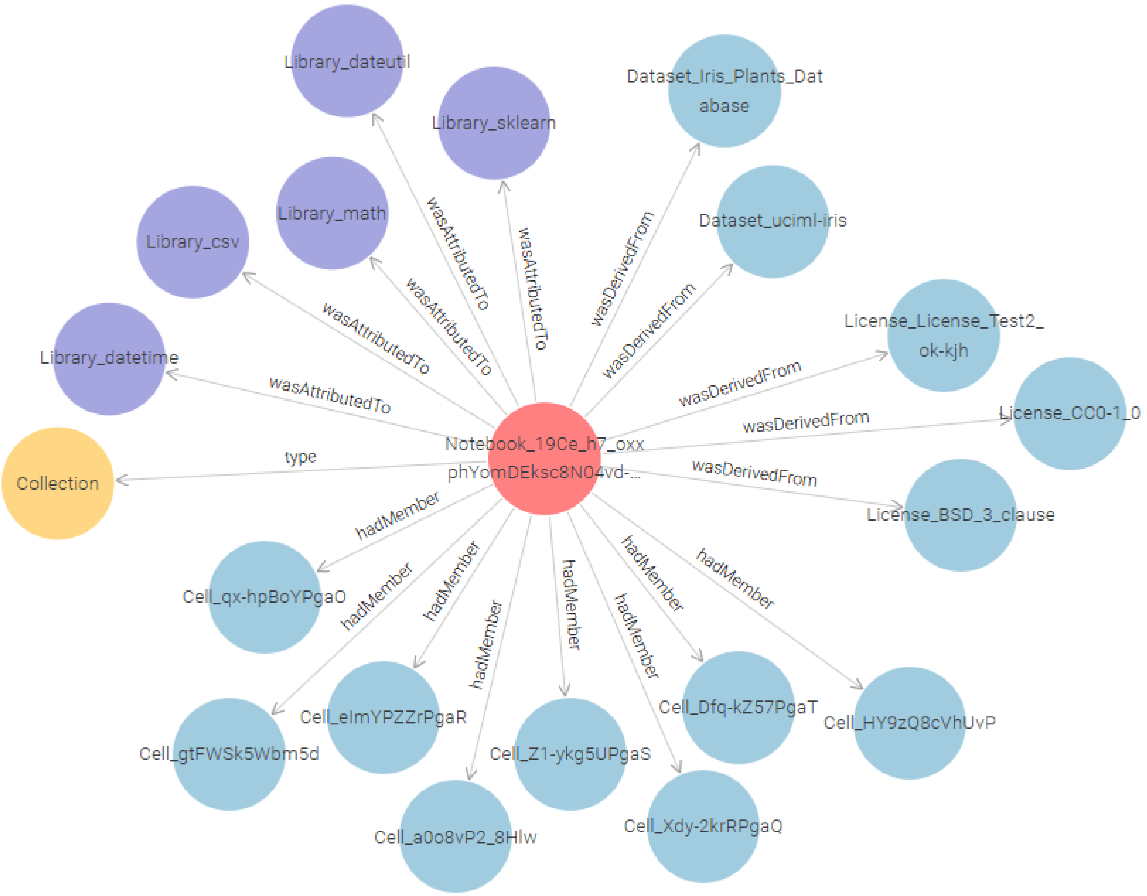


Figure 6.1: Visualization of a notebook and its properties in the knowledge graph with GraphDB. The notebook is of type `Collection` and is represented with its cells, libraries (*datetime*, *math*, etc.), datasets, and licenses.

The visualization tool allows noticing link between two notebooks, for example. Figure 6.2 illustrates the *math* library that is shared between two different notebooks. The visualization of the graph enables to illustrate interlinks.

As explained in Section 4.2, the licenses in the knowledge graph can be handled in various ways. They can be deduced from a Kaggle dataset using the Kaggle API or written in the notebook as a key-value pair: `# License: license_name`. Indeed, a license can be attributed to a dataset or to a piece of code. In Figure 6.3, the license is deduced from a Kaggle dataset, and in Figure 6.4, the license is written in the notebook. These two figures show the different ways a license can be represented in the knowledge graph. Figure 6.3 specifically visualizes license information related to datasets.

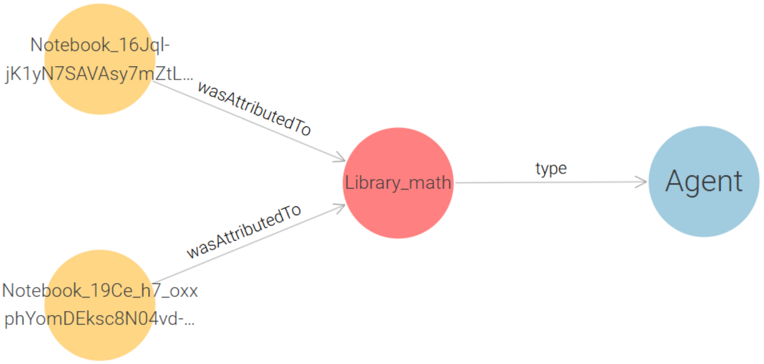

Notebook_16Jql-jK1yN7SAVAsy7mZtL...
wasAttributedTo
Library_math
type
Agent
wasAttributedTo
Notebook_19Ce_h7_oxx phYomDEksc8N04vd-...

Figure 6.2: Visualization of the *math* library shared between two different notebooks in the knowledge graph with GraphDB.

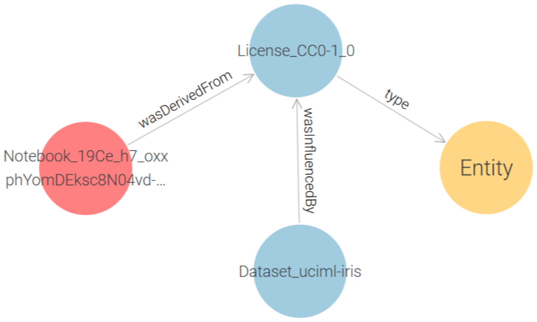


Figure 6.3: Visualization of a license deduced from a Kaggle dataset with GraphDB.

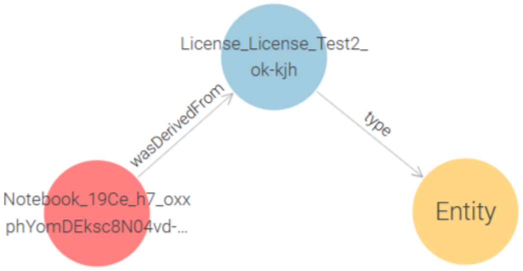

License_License_Test2_ok-kjh
wasDerivedFrom
type
Entity
Notebook_19Ce_h7_oxx phYomDEksc8N04vd-...

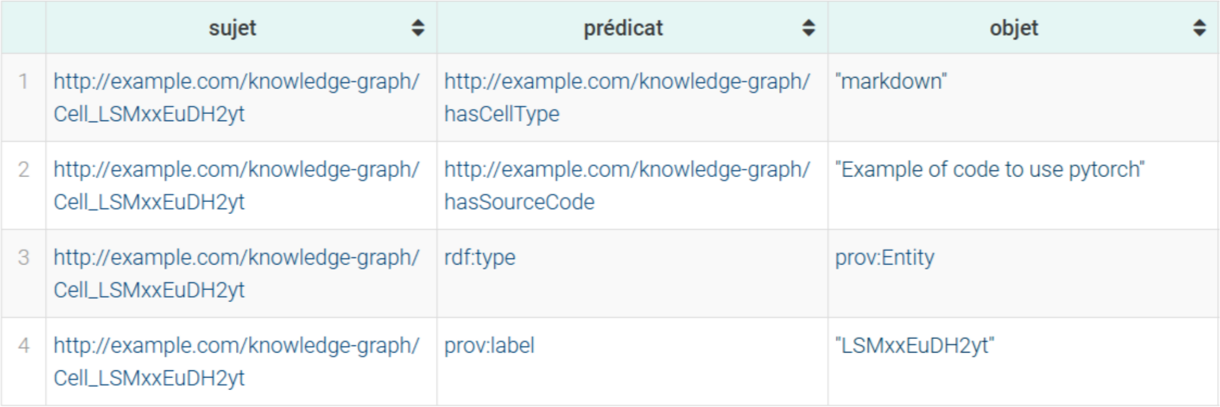
Figure 6.4: Visualization of a license as a comment in the notebook with GraphDB.

GraphDB also provides a tabular visualization of the RDF triples. A table of RDF triples with the cell *Cell_LSMxxEuDH2yt* as the subject is illustrated in Figure 6.5. This tabular visualization provides access to the properties of the selected cell, along with the objects associated with those properties.

In addition to checking the correctness of the constructed knowledge graph, the visualization helps to conceptually understand how the information is represented and how the different classes are interlinked together in the knowledge graph.

6.2 Basic SPARQL Queries for Graph Validation

To validate of the constructed knowledge graph, some simple queries have been executed using SPARQL. Several queries were written and tested on a knowledge graph built from two



	sujet	prédicat	objet
1	http://example.com/knowledge-graph/Cell_LSMxxEuDH2yt	http://example.com/knowledge-graph/hasCellType	"markdown"
2	http://example.com/knowledge-graph/Cell_LSMxxEuDH2yt	http://example.com/knowledge-graph/hasSourceCode	"Example of code to use pytorch"
3	http://example.com/knowledge-graph/Cell_LSMxxEuDH2yt	rdf:type	prov:Entity
4	http://example.com/knowledge-graph/Cell_LSMxxEuDH2yt	prov:label	"LSMxxEuDH2yt"

Figure 6.5: Table visualization of RDF triples associated with a specific cell in the knowledge graph with GraphDB.

test notebooks.

Among these queries, one aims at retrieving all the markdown cells. It returns the identifier for each cell and its corresponding notebook. This query is illustrated in Listing 6.1, and the result is shown in Table 6.1. The obtained results can be compared to the actual content of the notebook. In the two test notebooks, there are two markdown cells, one in each notebook, which is the result obtained from the query.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX myns: <http://example.com/knowledge-graph/>
PREFIX prov: <http://www.w3.org/ns/prov#>

SELECT ?Cell ?notebook WHERE {

    ?Cell myns:hasCellType "markdown" .
    ?notebook a prov:Collection .
    ?notebook prov:hadMember ?Cell.
}
```

Listing 6.1: SPARQL query that aims to get all the markdown cells and the notebook they belong to.

Many other queries have been tested and have returned the expected results. Among them were some simple queries, such as retrieving the libraries used in two different notebooks, and the licenses associated with a dataset. On the other hand, more complex queries were also considered. Those include retrieving identical cells or information about the author of a notebook, for example.

Table 6.1: Result of a SPARQL query applied to a knowledge graph based on two test notebooks. The query aims to get all the markdown cells and their corresponding notebooks.

Cell	**Notebook**
`http://example.com/knowledge-graph/Cell_LSMxxEuDH2yt`	`http://example.com/knowledge-graph/Notebook_16Jql-jK1yN7SAVAsy7mZtLBfn39D6gCB`
`http://example.com/knowledge-graph/Cell_a0o8vP2_8Hlw`	`http://example.com/knowledge-graph/Notebook_19Ce_h7_oxxphYomDEksc8N04vd-3RL9p`

6.3 SPARQL Query to Answer Questions about Data Lineage

Several challenges related to the data lineage were identified in Chapter 3. We aim to address these challenges by converting Jupyter notebooks into a knowledge graph and enabling queries with SPARQL.

In the introduction, the discussion around data lineage in notebooks led to various questions whose answers might help scientists in their research work. Among the given examples of questions, question Q3 relates to licenses and restrictions in a file. Listing 6.2 illustrates the SPARQL query to answer the part related to the licenses of this question. The query returns licenses and their corresponding notebooks. In addition, if the license is associated with a dataset, it also returns that dataset. Every license is related to a notebook. However, for the licenses deduced from a dataset through the Kaggle API, there is a second relationship with the dataset using the predicate: `prov:wasInfluencedBy`. Since not all licenses are linked to a dataset, the query comprises an OPTIONAL clause to find a dataset related to a license. Therefore, if no dataset is found, the query still returns the license, but only with the associated notebook.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX myns: <http://example.com/knowledge-graph/>

SELECT DISTINCT ?label_license ?label_notebook (IF(BOUND(?label_dataset), "
    Yes", "No") AS ?hasDataset) ?label_dataset
WHERE {
    # License
    ?license a prov:Entity.
    ?license prov:label ?label_license.

    # Notebook
    ?notebook a prov:Collection.
    ?notebook prov:label ?label_notebook.
    ?notebook prov:wasDerivedFrom ?license.
```

```
        FILTER(!regex(str(?license), "Dataset_"))

        OPTIONAL{
            # Dataset
            ?dataset a prov:Entity.
            ?dataset prov:label ?label_dataset.
            FILTER(regex(str(?dataset), "Dataset_"))

            ?notebook prov:wasDerivedFrom ?dataset.
            ?dataset prov:wasInfluencedBy ?license.
        }
    }
```

Listing 6.2: SPARQL query that aims to return the licenses associated with a notebook. If the license is retrieved from a dataset, it also returns that dataset in addition to the corresponding notebook.

The result of the query is represented in Table 6.2. The first two licenses are retrieved through a notebook; they were mentioned in comments in the code. The third one is a license that has been retrieved through the Kaggle public API. Therefore, it is associated with a notebook and a dataset: *Iris_Species*.

Table 6.2: Result of a SPARQL query applied to a knowledge graph based on two test notebooks. The query aims to retrieve the licenses associated with a notebook. If the license is retrieved from a dataset, it also returns that dataset in addition to the corresponding notebook.

License	**Notebook**	**From Dataset?**	**Dataset**
BSD_3_clause	19Ce_h7_oxxphYomDEksc8N04vd-3RL9p	No	None
License_Test2_ok-kjh	19Ce_h7_oxxphYomDEksc8N04vd-3RL9p	No	None
CC0-1.0	19Ce_h7_oxxphYomDEksc8N04vd-3RL9p	Yes	Iris_Species

This query helped to retrieve important information about the notebook without the need to look at the code. It is even more helpful when the license is related to a Kaggle dataset. Indeed, it is through external data that the information is accessible, thus making it less easily accessible to find the information by hand. This query is an example of the kind of query that can help provide easy access to specific information.

Other queries can be designed to answer questions related to the faced challenges. For example, as the knowledge graph contains the source code of each cell of a notebook, it is thus possible to retrieve identical cells in one or more notebook and detect duplicate cells.

6.4 Exploration of the KGTorrent Dataset

In the introduction, we have mentioned the significant role of Jupyter notebooks in the field of data science. In this thesis, we focus primarily on analyzing these files, as they serve as the main source of data. To carry out the desired analyses, access to a real-world dataset of Jupyter notebooks is essential. Once the basic case is considered with a few files that have been visualized and queried, this dataset is used to designed analyses under real-world conditions.

The KGTorrent dataset is a collection of Python Jupyter Notebooks proposed by Luigi Quanranta et al. in [33]. This dataset is composed of 248.761 publicly available Python Jupyter Notebooks. The authors have collected the data from Kaggle[1], a platform dedicated to data science and machine learning. The dataset offers a large variety of notebooks, each enriched with rich metadata and content. The main objective of the KGTorrent dataset is to provide a large collection of data to analyze the content of Jupyter notebooks. Additionally, it aims to identify potential weaknesses in these notebooks for future extensions.

Moreover, this dataset has already let several works highlight the importance of code quality and style in Jupyter Notebooks. It has been used to analyze how data scientists use notebooks and how these notebooks are typically used in data science workflows [33]. Md Saeed Siddik et al. have used the KGTorrent dataset to answer the question: *Do Code Quality and Style Issues Differ Across (Non-)Machine Learning Notebooks?* [3]. The authors analyze whether the code quality is related to the use of machine learning. With the help of the KGTorrent dataset, they have found that the code quality is hugely different between machine learning and non-machine learning notebooks.

In this thesis, we leverage the KGTorrent dataset to construct a knowledge graph and then query it. The following section illustrate these two stages.

6.5 Knowledge Graph with a Realistic Dataset

Thanks to the KGTorrent Dataset from Kaggle, which is a large dataset, the construction of a knowledge graph can be simulated in real-world conditions. This dataset contains various types of notebooks, as explained in Section 4.1.1. The focus as been made on notebooks created and executed on Google Colab to ease the process. Therefore, all the notebooks in the dataset were filtered to keep only the desired ones. The remaining number of notebooks went from 248.761 to 3.536. It means that only 7% of the initial dataset was built from Google Colab.

Among the 3.536 filtered notebooks, 100 notebooks are then randomly selected to build

[1] `https://www.kaggle.com/`

the knowledge graph. 10% of those notebooks were manually enriched with key-value pairs to include more metadata. Indeed, those notebooks do not follow the convention explained in Section 4.3, which specifies how to comment the code to add information about the notebook. In the subsections, the modified dataset composed of the 100 notebooks is called the *small dataset*.

6.5.1 Performance Measurements to Generate the Knowledge Graph

To generate the knowledge graph, all the notebooks have to be transformed into RDF triples through the mapping rules. This operation might take time, depending on the amount of information that has to be processed. We have observed with the two tests notebooks that the time increases if a connection with an external source is required. For example, when a dataset is mentioned as originating from Kaggle, the time to generate the RDF triples increases. Indeed, a connection to the Kaggle public API is thus made to retrieve data related to that dataset.

Two points of view have been considered to analyze the performance measurements of generating a knowledge graph constructed on the small dataset. First, the generation time of RDF triples for each notebook is represented using a histogram illustrated in Figure 6.6. The histogram illustrates a right-skewed distribution of the generation time. It means that most of the notebooks are generated in a short time (less than 5 seconds), while the others take significantly longer to be generated (long tail). The mean time of 8 seconds arises slightly after the peak of the histogram, which is consistent with the observed skewness.

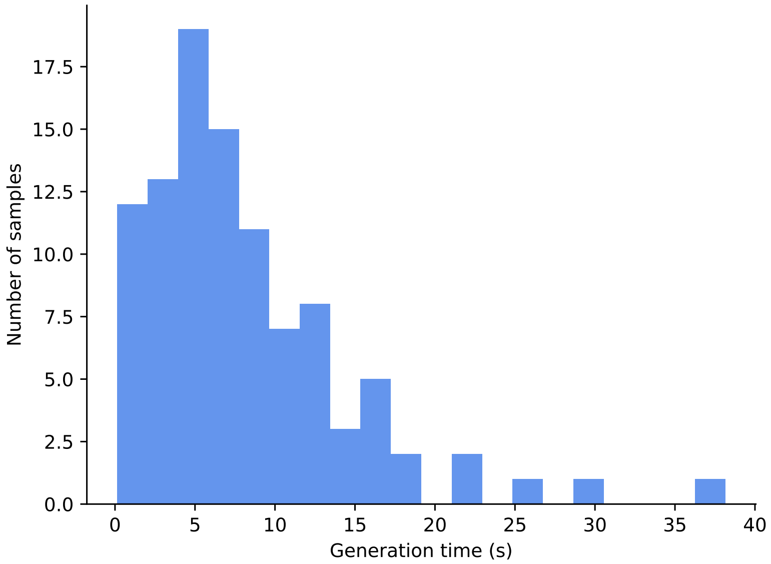


Figure 6.6: Histogram of the time needed for each notebook of the small dataset to generate their RDF triples.

Building upon the previous analysis, which exhibits a right-skewed distribution, Figure 6.7 provides a more granular view. It illustrates the time required for each notebook to generate their RDF triples, along with the mean and the standard deviation time. Additionally, it shows that the time required to generate the triples might vary from one notebook to another; some notebooks take less than 1 second to generate the triples, while others take over 30 seconds.

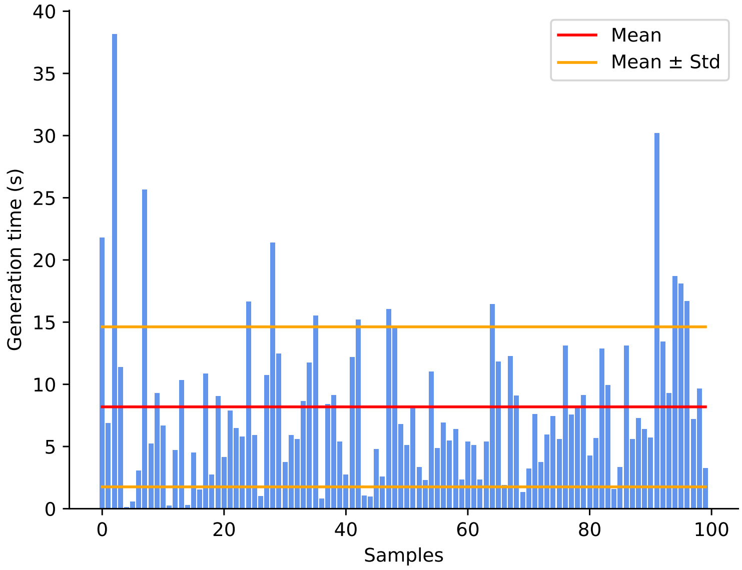


Figure 6.7: Representation of the time required for each notebook of the small dataset to generate its RDF triples.

The performance measurements analyzed in Figures 6.6 and 6.7 let us consider that the time required to generate the knowledge graph for the small dataset is acceptable, although some notebooks might take more time.

Nevertheless, we must keep in mind that only 10% of the notebooks in the small dataset have been modified. If more notebooks are modified, or if all the notebooks follow the convention of annotation and thus contain more information to process, the time required to generate the knowledge graph might increase.

To gain further insight about the variation in generation time between notebooks, Table 6.3 is generated. It corresponds to ten notebooks with the highest generation time of RDF triples. The table also includes the file size and the number of imports for each notebook. In this table, three of the ten notebooks are modified, and all are in the top four slowest notebooks.

First, the size of the notebook is analyzed and compared with the generation time of RDF triples. As some notebooks in the small dataset are huge and do not appear in Table 6.3, while others are smaller and do appear, it might seem that there is no relationship between the size of the notebook and the generation time. The correlation between these two variables is computed and corroborates this assumption, as the correlation coefficient is close to 0.

Table 6.3: Ten notebooks with the highest generation time of RDF triples from the small dataset. The size of each notebook and the number of imports are also presented. The notebook with the biggest generation time was modified and contains a Kaggle dataset, which is retrieved through the Kaggle API.

File Index	Time [s]	Modified	Size [ko]	Number of Import
2	38.15	yes	1.226	69
91	30.22	no	89	68
7	25.65	yes	4.123	49
0	21.79	yes	170	48
28	21.42	no	2.488	38
94	18.72	no	54	33
95	18.11	no	47	25
96	16.70	no	195	33
24	16.66	no	216	28
64	16.47	no	71	30

With the two test notebooks, we have observed that the generation time of RDF triples increased when connections with external sources are required. This might be the case for the notebooks in Table 6.3. There are three possible reasons to initiate the connection to one of the three public APIs considered in this thesis: a Kaggle dataset is mentioned, an import is made, or a GitHub account is specified. Among the 10% of the modified notebooks, we have added three Kaggle datasets to three different notebooks and no GitHub account. Therefore, the most significant amount of connection is caused by the number of imports. Indeed, each imported library requires a connection to the PyPI API to retrieve the metadata of the library. For example, the slowest notebook in Table 6.3 contains 69 imports, which leads to 69 connections to the PyPI API in addition to one connection to the Kaggle API to retrieve the dataset.

For every notebook in the small dataset, we have counted the number of imports to analyze it in relation to the generation time. The correlation coefficient between the number of imports and the generation time of RDF triples is around 0.96, which means that there is a strong correlation between the number of imports and the generation time. Figure 6.8 illustrates the relationship between these variables for each notebook of the small dataset. From this, we can assume that the connection to external sources is the reason why the generation time of RDF triples is higher for some notebooks.

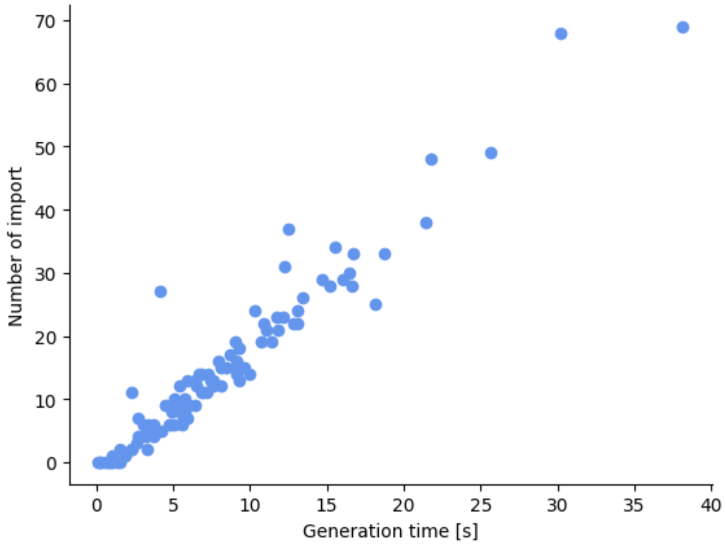


Figure 6.8: Illustration of the relationship between the generation time of RDF triples and the number of imports for each notebook of the small dataset. The correlation coefficient is around 0.96.

6.5.2 Simple Queries with the Realistic Dataset

The knowledge graph based on the small dataset is operable and can thus be queried. This graph contains more RDF triples; it is thus interesting to measure the execution time of queries that are performed. The SPARQL queries are executed on the knowledge graph based on the small dataset using *Apache Jenna Fuseki.* It acts as a SPARQL server that enables the retrieval of data from the knowledge graph through SPARQL queries. It also provides the SPARQL Graph Store protocol [41]. This lets us query the knowledge graph we have built with the small dataset with a dedicated tool. Multiple queries have been applied to the knowledge graph, but only two queries are presented in this section. The first one is a complex query, while the second is less complex and executed faster.

Consider that we want to have access to the number of libraries that are shared between two different notebooks. The query that answers this question is illustrated in Listing 6.3, and the result is in Table 6.4. The execution time of this query is around 0.339 seconds, and the results are composed of more than 3500 notebook pairs.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX prov: <http://www.w3.org/ns/prov#>


SELECT DISTINCT (COUNT(?label_library1) AS ?count) ?label_notebook1
    ?label_notebook2
WHERE {
    ?notebook1 a prov:Collection.
    ?notebook1 prov:label ?label_notebook1.
    ?notebook2 a prov:Collection.
    ?notebook2 prov:label ?label_notebook2.

    ?library1 a prov:Agent.
    ?library2 a prov:Agent.

    ?notebook1 prov:wasAttributedTo ?library1.
    ?notebook2 prov:wasAttributedTo ?library2.
    ?library1 prov:label ?label_library1.

    FILTER (STR(?notebook1) < STR(?notebook2) && ?library1 = ?library2).
}
GROUP BY ?label_notebook1 ?label_notebook2
```

Listing 6.3: SPARQL query that returns the number of libraries shared between two different notebooks.

Table 6.4: Result of a SPARQL query applied to a knowledge graph built upon the small dataset. The query aims at returning the number of libraries shared between two different notebooks.

Notebook 1	**Notebook 2**	**Nb Libraries**
BvmocDG28ClAd2mvN5sO	XsYlYy5pCwAJYg7XyiqY	6
BvmocDG28ClAd2mvN5sO	nHnYKVuT4CuSR1LeJWDX	10
EEKPxhv9VCLQK5SoSI05	Ew6bwQ6H06CrnO8l8roZ	2
fYAJe8odeBBQ1p5eOW4O	uiXTyhe4ByHRTOcT0ecJ	6
XsYlYy5pCwAJYg7XyiqY	sILgX1sZ8IY2epTTSJdN	5
Gsz1G0PCnIDdzvNxuDIh	nHnYKVuT4CuSR1LeJWDX	6
2VIjhTvZV46Em86wuc0f	EtowdzXJDmlihxhQ9RB4	7
...	..	..
WnRc1PHODvua1wHoi1Zd	uiXTyhe4ByHRTOcT0ecJ	1
WnRc1PHODvua1wHoi1Zd	x7bln8bN5zl2aP1VcK2E	1

A second query, which aims at returning all the defined functions and their corresponding notebooks, is illustrated in Listing 6.4. This query is less complex and fast compared to the first one, with an execution time of 0.03 seconds. In the results, the overall number of defined functions is around 300, and some are represented in Table 6.5.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX prov: <http://www.w3.org/ns/prov#>
PREFIX myns: <http://example.com/knowledge-graph/>

SELECT DISTINCT ?functions ?label_notebook WHERE {
    ?notebook a prov:Collection.
    ?notebook prov:label ?label_notebook.
```

```
        ?notebook myns:hasDefinedFunction ?functions.
    }
```

Listing 6.4: SPARQL query, which aims to retrieve all the defined functions in each notebook.

Table 6.5: Result of a SPARQL query applied to a knowledge graph built upon the small dataset. The query aims to retrieve all the defined functions and their corresponding notebooks.

Function	**Notebook**
create_np_array_from_input_list	2VIjhTvZV46Em86wuc0f
make_decision_tree	2VIjhTvZV46Em86wuc0f
check_keywords	2tAPlqe4Ny9e0bEx27bD
...	..
backwardelimination	wTkiUIrs3NXbC5OM4IhI
build_model	wTkiUIrs3NXbC5OM4IhI

The two above queries show that a realistic knowledge graph built on a small dataset can be queried and return answers to various questions. Additionally, the execution time seems dependent on the complexity of the query and the amount of processed data. The first query is complex, deals with a lot of data, and takes 10 times longer than the second to be executed.

6.6 Summary of the Evaluation and Experimentation

This chapter aims at evaluating the structure of the knowledge graph and experimenting the approach with complex queries, as well as with a realistic dataset.

The evaluation was realized with a knowledge graph built upon two test notebooks to check the correctness of the designed and implemented structure. This was done through the visualization of the graph with GraphDB and the execution of several simple SPARQL queries. These two steps have led to the conclusion that the graph structure is correctly built and represents the information contained in the notebooks as expected by the design of the structure.

After the validation of the structure, it comes the querying phase of the knowledge graph related to data lineage challenges. A complex query related to licenses shows the expected results and proves that the knowledge graph can be useful to face these challenges.

Finally, we investigated and build a knowledge graph on a realistic dataset: a dataset of 100 notebooks randomly selected from the KGTorrent dataset. Performance measurements to build the knowledge graph were analyzed and have shown that the generation time of RDF triples is dependent on the number of connections with a public API. Queries were then applied to this knowledge graph, and the execution time of each query was analyzed. We observed that the more complex the query, the more time it takes to execute it. However, the execution times are acceptable for both simple and complex queries.

Chapter 7

Discussions

This thesis aims to present a knowledge graph-based approach to solving the challenges of tracing data lineage in data science. We constructed a structured approach using provenance data, which has then been evaluated. However, through the entire process of the construction of the approach, several challenges were faced, and some limitations were identified. Additionally, certain choices were made, such as the use of notebooks created and executed with Google Colab for the creation of the knowledge graph. This chapter presents a discussion of the results, the limitations faced, and the constraining choices made.

7.1 A Declarative Approach to Knowledge Graph Generation

The main limitation faced during the construction of the knowledge graph was the language used to map the data. Initially, we wanted to use RML, which is a declarative language that enables the mapping of data from heterogeneous sources. This choice was for several reasons detailed in Section 5.2. However, two issues were faced during the construction of the knowledge graph that led us to use an imperative language: Python.

The first issue is the lack of extensions for the RMLMapper tool. Indeed, we needed to be able to access and extract the desired information directly from the source code of each cell. However, this tool does not allow for the extraction of the desired data to populate the knowledge graph effectively. An alternative was Morph-KGC. This tool enables the extraction of the desired data from the source code of the cells.

The second issue is the iterator problems in the RML language. When a notebook is created, the whole JSON file is considered as the notebook. However, the unique identifier of the notebook is specified in one of the cells by following the convention of annotation explained in Section 4.3. With RML, when we get access to the identifier of the notebook, the accessible information is reduced to that cell. All the other cells and other information contained in the

JSON file are no longer accessible. Essentially, the link between the cells and the notebook itself was broken, except for one cell. Furthermore, the limited join capabilities of RML were insufficient for our needs. We required a theta-join to set relationships between each cell and the notebook itself.

We must mention that RML is a language that is not yet standardized and is thus in development. The issues we faced could be solved in the future with the evolution of the language, as some research is currently done related to this language and its issues. For example, BURP [42] is a reference implementation of the RML language, which supports the new RML modules and has shown several problems related to them.

7.2 Generalization to a Uniform Notebook Format

In this work, we decided to focus on a specific type of notebook: the Google Colab notebooks. Indeed, there are different types, each with a different format of key-value pair When we consider two different formats, the bodies share the same idea, but the names of the keys can be different, or the same keys can be present at two different levels of the JSON file. This consideration is important, as the approach presented in this work is not generalizable to all types of notebooks. This has been noticed during the evaluation of a real-world dataset in Section 6.5, where the majority of notebooks are not in the Google Colab format.

Moreover, the metadata fields in the notebooks are often limited. To consider more data, we have defined a structure to extract information from the source code of the cells, as explained in Section 4.3. However, this convention is strict and is not followed by the majority of the notebooks. Therefore, information about authors, dates, or datasets cannot be easily collected. This is a binding choice that has been made to be able to collect the information we need to build the knowledge graph.

Additionally, the presented structure led to manual modifications of some notebooks from the KGTorrent dataset for the experimentation phase. This intervention underscores the limitation related to the generalization of the approach presented in this work. It also highlights the challenges that we faced during the construction of the knowledge graph: the lack of a standardized structure for all the different types of notebooks and the few metadata fields.

7.3 Extension of the PROV-O Ontology

The PROV-O ontology is considered as the basis of this work to represent the provenance data. However, this ontology is not enough to represent all the information that we want to store in the knowledge graph. Therefore, it was extended with the FOAF vocabulary to represent the authors of the notebooks in a more detailed way. In addition, some new terms were defined to describe data that was too specific to be represented with the PROV-O ontology or the FOAF vocabulary. For example, the predicate `hasSourceCode` has been defined to represent the source code of a cell.

Approximately ten predicates have then been created, all with a range of `string`. However, some of these terms could have been defined with a more specific range, such as `hasCellType`, which could have a range restricted to `Code` or `Markdown`. This would have been more precise to describe the type of cell in a notebook. The same could have been done for the `hasOutputType` term, which could have a range restricted to `stream` or `display`.

7.4 Deployment of the Knowledge Graph

The structure of the knowledge graph has been designed to represent a notebook and its content. As provenance data is the main source of information, the PROV-O ontology was considered. To complete the ontology, the FOAF vocabulary and additional created terms have been used. Thus, a namespace was created to represent the combination of PROV-O, the FOAF vocabulary and the new terms. `http://example.com/knowledge-graph/` is the namespace used in this thesis, where `example.com` is a usual fictive placeholder in the context of a research work, and `knowledge-graph` is here to specify the context of the namespace. Note that it is a fictive namespace and it is not registered. This namespace is thus dedicated to the research field and cannot be used in a real-world application.

The deployment of the ontology needs to be done in a real-world application to be able to use the knowledge graph in a real-world context. This would need to ask for a persistent identifier from the W3id service, which is a service that provides persistent identifiers for the Web [43]. However, these identifiers are granted only for "serious" projects and not for projects from students or Master thesis. In the future, the deployment of the ontology and the knowledge graph could be used in the field of data science to answer questions related to data lineage and provenance data.

7.5 Blank Nodes in the Construction of the Graph

In the construction of the knowledge graph, some constraining choices have been made. One of them is related to the use of blank nodes to represent the authors of notebooks. Indeed, the name of the author of a notebook is an information that can be found in the source code of a cell by following the comment convention presented in Section 4.3.

During the construction of the knowledge graph, we faced an issue related to the representation of the authors of notebooks. Indeed, the name of an author might be contained in the source code of a cell, but this information is not sufficient to create a unique identifier for each author, as multiple authors can have the same name. Therefore, we had to store the author as a blank node in the knowledge graph. However, the consequence of this choice is that if two notebooks have the same author name, these authors will be considered as the same person in the graph even if they are two different persons.

This is a limitation of the approach because, if two different researchers have the same name, these cannot be distinguished in the knowledge graph. A solution to this issue would be to use the GitHub account of the author, as a unique identifier; this information is already extracted from the source code of a cell and stored as triples. However, the latter is not considered a unique identifier as we suppose that not all the authors have a GitHub account. Additionally, this information is not always present in the source code of a cell.

7.6 Exploring Evaluation and Experimentation Results

After defining mapping rules to match the structure established in the design of the knowledge graph, the construction of a knowledge graph with data from two test notebooks has been done. This initial graph was evaluated and queried to ensure its validity. Then some experiments with a knowledge graph built upon a real-world dataset were made.

7.6.1 Evaluation of the Knowledge Graph

The results of the evaluation have proved the correctness of the defined mapping rules, leading to a valid knowledge graph; the representation of the data stored in the knowledge graph is consistent with the design of the graph. This evaluation has been done with the visualization of the graph and with the use of SPARQL queries.

This evaluation is crucial, as it enables highlighting if there are any programming mistakes or misunderstandings. Indeed, it is easy to make spelling mistakes in the name of the predicates while defining the mapping rules (`wasAttributedto` instead of `wasAttributedTo`) or forget to define a `label` for a given class, for example.

7.6.2 Addressing Data Lineage Challenges

The graph, composed of triples from the two test notebooks, was queried with a SPARQL query that was designed to answer a question related to the data lineage challenges. The question was about licenses that are associated with a notebook through source code or datasets. The query lets a user have easy access to those licenses, even if they are not explicitly mentioned in the notebook (retrieved from a dataset through the Kaggle API). This query, which takes less than a second to be executed, provides quick access to some information that can be useful in the context of data lineage.

However, there might be questions that we would like to ask, but that cannot be answered with the current structure of the knowledge graph. For example, if we want to have access to transformations that were applied to a dataset, we would need to consider more information and think of its representation in the structure of the knowledge graph. Of course, some snippets of information that can answer a part of that question are already present in the graph. All the functions called in the code cells are stored in the graph; in those, we can find, for example, scaling functions, exponential or logarithmic functions, etc. With this information, we can have an idea of the transformations that have been applied to a dataset. We can also have information about the machine learning model that has been used in the notebook, or which metric is used to evaluate the model.

To address other questions, the graph needs to be enriched. This requires thinking about the representation of the information, the consideration of valuable information, and the extraction of this information. Moreover, it would be interesting to explore other external sources of information.

7.6.3 Experimentation on a Real-world Dataset

In the experimentation phase on a real-world dataset, some queries were executed on a knowledge graph built on 100 notebooks from the KGTorrent dataset. Additionally, performance measurements were considered to evaluate the generation time of the knowledge graph and the performances of the queries.

With some statistical analysis of the generation time of RDF triples, we have observed that some notebooks need more time than others to generate the triples. The size of the notebook file was first investigated as a possible explanation for this generational difference. However, no relationship was found between the size of each notebook file and the generation time. Then, we have considered the number of accesses to APIs, as a request has to be sent to an external server every time a library is imported, a Kaggle dataset is specified, or a GitHub account is mentioned. This consideration has led to the discovery of a relationship between the number of imported libraries, which translates to the number of accesses to the PyPI API, and the generation time of RDF triples for each notebook.

Several queries were then executed on the knowledge graph. The execution time was studied to evaluate the efficiency of the queries in real-world conditions. We have observed that some complex queries have a longer execution time compared to simple ones. Nevertheless, the time of execution is under a second for all the tested queries, which is acceptable for a real-world application.

The experimentation considered one hundred notebooks, of which only 10% were modified to follow the convention imposed to extract data from the source code of the cells. Therefore, if all the notebooks follow the convention, more information might be extracted, and it might take longer to generate the graph and query it.

7.7 Summary of Discussions

This chapter explored the limitations, the binding choices, and the results of our work. Key areas for potential enhancement include the enrichment of the graph to address other questions related to the data lineage challenges and the RML mapping language. These considerations lead us to the conclusion in the next chapter.

Chapter 8

Conclusions

Data science is a field that uses data to extract knowledge. However, tracking the origin and transformations of this data (data lineage) often leads to challenges. Jupyter notebooks, stored as JSON files, are frequently used tools in data science. These allow the writing of code, data visualizing, the documentation of the work, and the sharing of results in a single document. This thesis focuses on the challenges related to data lineage in Jupyter notebooks; specifically the aim is to present an approach to overcoming challenges related to data lineage using knowledge graphs and provenance data.

To address data lineage challenges, we designed an approach that first build a structure that represents the information extracted from notebooks in the form of a graph. In this way the data sources, including notebooks generated and executed on Google Colab, along with external sources like public APIs were analyzed. The purpose was to understand the data and represent it in a graph with the most appropriate structure. Since the JSON format of a notebook varies depending on the tool used to generate and execute it, only notebooks built from Google Colab are considered to ensure consistency. The three APIs that were used to enrich the notebooks are PyPi API, GitHub REST API, and Kaggle API. To design the structure of the data the PROV-O ontology was chosen as a foundation, extended with the FOAF vocabulary and some newly defined terms to globally represent the information.

Once the structure was well-defined, the data from the data sources was mapped to the ontology using mapping rules. Those rules were initially defined in RML, but this language has some limitations which make RML not suitable in the context of this research. Therefore, we coded the RDF Generation process imperatively in Python using RDFLib, as it overcomes the limitations of RML.

The approach was then validated by checking the correctness of a generated knowledge graph on a small and non-realistic dataset composed of two test notebooks. A visualization tool called GraphDB was used to visualize the graph and check for any errors in the data that could be found visually. Additionally, several queries were executed on that graph to answer questions about all the information that was collected and represented in the graph. These queries successfully retrieved the expected results and thus demonstrated the ability to correctly represent the data in the knowledge graph.

Finally, some experimentation was done to assess whether the approach was able to answer questions about data lineage challenges and if the approach is viable in the context of a real-world dataset. A complex query related to the access of licenses associated with a notebook or a dataset was designed and demonstrated the ability of the knowledge graph to answer a question in the context of data lineage. Then, more experimentation was performed on a knowledge graph built upon a dataset composed of one hundred notebooks. The generation time of that graph was analyzed, and it was proven that the number of connections to an API for one notebook is related to the generation time of RDF triples for that notebook. Additionally, several queries were executed on that knowledge graph, resulting in the expected results and obtained in an acceptable amount of time.

In conclusion, the use of knowledge graphs and data provenance to overcome challenges related to data lineage in the case of Jupyter notebooks has proven to be a promising approach. Indeed, the presented results highlight that the approach enables the answering of specific questions that help to provide access to information without the need to read the whole notebook or to perform manual research. In addition, the time to answer questions or generate a knowledge graph on more than one hundred notebooks is short enough to be used in real-world conditions. From this work, we have learned that notebooks can be represented semantically with a knowledge graph, that public APIs related to information contained in notebooks can enrich the knowledge graph, and that the graph can be queried to answer specific data lineage questions. However, the approach is still a prototype, and some improvements in the design of the ontology might help to answer more questions related to data lineage, and work on the mapping rules might allow the use of any type of notebook.

8.1 Future Work

This work has led to promising results in the mapping of notebooks to a knowledge graph and the leveraging of SPARQL queries to address data lineage challenges. However, it also opens the door to opportunities for further improvement. Indeed, there are two main limitations that have been identified, and it might be interesting to improve them in the future.

In data integration, the declarative language RML and the limitations associated with it encountered in this work constitute a possible future work. It would be interesting to investigate how this language can be improved to allow better data representation.

In data science, the heterogeneity of the JSON format of Jupyter notebooks does not allow adequate sharing of data or the use of data from various sources in one context. It might be interesting to explore the reasons of this heterogeneity and investigate the feasibility of developing a standard format for Jupyter notebooks. This standardization might improve interoperability and would highly enhance this work.

Bibliography

[1] Robert Ikeda and Jennifer Widom. Data lineage: A survey. Stanford University, 2009. `https://api.semanticscholar.org/CorpusID:7079031`.

[2] Oracle. Json defined. `https://www.oracle.com/uk/database/what-is-json/`. Accessed 05-04-2024.

[3] Md Saeed Siddik and Cor-Paul Bezemer. Do code quality and style issues differ across (non-)machine learning notebooks? yes! In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 72–83, 2023.

[4] Khairul Alam and Banani Roy. Challenges of provenance in scientific workflow management systems. In *2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 10–18, 2022.

[5] Hassan Hussein, Kheir Eddine Farfar, Allard Oelen, Oliver Karras, and Sören Auer. Increasing reproducibility in science by interlinking semantic artifact descriptions in a knowledge graph. In Dion H. Goh, Shu-Jiun Chen, and Suppawong Tuarob, editors, *Leveraging Generative Intelligence in Digital Libraries: Towards Human-Machine Collaboration*, pages 220–229, Singapore, 2023. Springer Nature Singapore.

[6] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 1345–1350, New York, NY, USA, 2008. Association for Computing Machinery.

[7] Timothy Lebo, Satya Sahoo, Deborah McGuinness, Khalid Belhajjame, James Cheney, David Corsar, Daniel Garijo, Stian Soiland-Reyes, Stephan Zednik, and Jun Zhao. *PROV-O: The PROV Ontology*. W3C Recommendation. World Wide Web Consortium, United States, April 2013.

[8] Eric Prud'hommeaux, Steve Harris, and Andy Seaborne. *SPARQL 1.1 Query Language*. W3C Recommendation. World Wide Web Consortium, March 2013.

[9] Guus Schreiber and Yves Raimond. *RDF 1.1 Primer.* W3C Recommendation. World Wide Web Consortium, June 2014.

[10] Olaf Hartig, Pierre-Antoine Champin, Gregg Kellogg, and Andy Seaborne. *RDF 1.2 Concepts and Abstract Syntax.* W3C Recommendation. World Wide Web Consortium, March 2024.

[11] W3C OWL Working Group. *OWL 2 Web Ontology Language Document Overview (Second Edition).* W3C Recommendation. World Wide Web Consortium, December 2012.

[12] Dan Brickley and Libby Miller. *FOAF Vocabulary Specification 0.99.* RDF and Semantic Web developer community, January 2014.

[13] Christophe Debruyne. Knowledge representation and reasoning lecture 2: Knowledge graphs, 2023.

[14] DBpedia Community. Global and unified access to knowledge graphs. `https://www.dbpedia.org/`. Accessed: 12-04-2024.

[15] Google. Api google knowledge graph search. `https://developers.google.com/knowledge-graph?hl=fr`. Accessed: 12-04-2024.

[16] IBM. Qu'est-ce qu'un graphe de connaissances ? `https://www.ibm.com/fr-fr/topics/knowledge-graph`. Accessed 05-04-2024.

[17] Souripriya Das, Seema Sundara, and Richard Cyganiak. *R2RML: RDB to RDF Mapping Language.* W3C Recommendation. World Wide Web Consortium, September 2012.

[18] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. RML: a generic language for integrated RDF mappings of heterogeneous data. In Christian Bizer, Tom Heath, Sören Auer, and Tim Berners-Lee, editors, *Proceedings of the 7th Workshop on Linked Data on the Web*, volume 1184 of *CEUR Workshop Proceedings*, April 2014.

[19] Ana Iglesias-Molina, Dylan Van Assche, Julián Arenas-Guerrero, Ben De Meester, Christophe Debruyne, Samaneh Jozashoori, Pano Maria, Franck Michel, David Chaves-Fraga, and Anastasia Dimou. The rml ontology: A community-driven modular redesign after a decade of experience in mapping heterogeneous data to rdf. In Terry R. Payne, Valentina Presutti, Guilin Qi, María Poveda-Villalón, Giorgos Stoilos, Laura Hollink, Zoi Kaoudi, Gong Cheng, and Juanzi Li, editors, *The Semantic Web – ISWC 2023*, pages 152–175, Cham, 2023. Springer Nature Switzerland.

[20] RML.io. RMLMapper. `https://github.com/RMLio/rmlmapper-java`, accessed: 2024-04-02.

[21] De Meester, Ben and Jozashoori, Samaneh and Maria, Pano and Chaves-Fraga, David and Dimou, Anastasia. *RML-FNML*. Knowledge Graph Construction Community Group, March 2024. `https://kg-construct.github.io/rml-fnml/spec/docs/`.

[22] Julián Arenas-Guerrero, David Chaves-Fraga, Jhon Toledo, María S. Pérez, and Oscar Corcho. Morph-KGC: Scalable knowledge graph materialization with mapping partitions. *Semantic Web*, 15(1):1–20, 2024.

[23] Amazon Web Services. What is data science? `https://aws.amazon.com/fr/what-is/data-science`. Accessed 19-05-2024.

[24] Paramita (Guha) Ghosh. Data management vs. data science. `https://www.dataversity.net/data-management-vs-data-science/`, Mar 2022. Accessd 19-05-2024.

[25] Andreas P. Koenzen, Neil A. Ernst, and Margaret-Anne D. Storey. Code duplication and reuse in jupyter notebooks. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9, 2020.

[26] Miguel-Ángel Sicilia, Elena García-Barriocanal, Salvador Sánchez-Alonso, Marçal Mora-Cantallops, and Juan-José Cuadrado. Ontologies for data science: On its application to data pipelines. In Emmanouel Garoufallou, Fabio Sartori, Rania Siatri, and Marios Zervas, editors, *Metadata and Semantic Research*, pages 169–180, Cham, 2019. Springer International Publishing.

[27] Mark Schildhauer, Matthew B. Jones, Shawn Bowers, Joshua Madin, Sergeui Krivov, Deana Pennington, Ferdinando Villa, Benjamin Leinfelder, Christopher Jones, and Margaret O'Brien. Oboe: the extensible observation ontology, version 1.2. KNB Data Repository, 2016.

[28] J. Zheng, M.R. Harris, and A.M. et al. Masci. The ontology of biological and clinical statistics (obcs) for standardized and reproducible statistical analysis. *Biomed Semant*, 7:53, 2016.

[29] Sheeba Samuel and Birgitta König-Ries. Provbook: Provenance-based semantic enrichment of interactive notebooks for reproducibility. Heinz-Nixdorf Chair for Distributed Information Systems Friedrich-Schiller University, Jena, Germany, 2018.

[30] Daniel Garijo and Yolanda Gil. Augmenting prov with plans in p-plan: Scientific processes as linked data. In *LISC@ISWC*, 2012.

[31] Shivani Choudhary, Tarun Luthra, Ashima Mittal, and Rajat Singh. A survey of knowledge graph embedding and their applications. *ArXiv*, abs/2107.07842, 2021.

[32] Ioannis Dasoulas, Duo Yang, and Anastasia Dimou. Mlsea: A semantic layer for discoverable machine learning. In Albert Meroño Peñuela, Anastasia Dimou, Raphaël Troncy, Olaf Hartig, Maribel Acosta, Mehwish Alam, Heiko Paulheim, and Pasquale Lisena, editors, *The Semantic Web*, pages 178–198, Cham, 2024. Springer Nature Switzerland.

[33] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. Kgtorrent: A dataset of python jupyter notebooks from kaggle. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2021.

[34] PyPi. Json api. `https://warehouse.pypa.io/api-reference/json.html`. Accessed: 20-03-2024.

[35] GitHub. Rest api endpoints for users. `https://docs.github.com/en/rest/users/users?apiVersion=2022-11-28`. Accessed: 20-03-2024.

[36] Kaggle. How to use kaggle. `https://www.kaggle.com/docs/api`. Accessed: 04-04-2024.

[37] codefresh by Octopus Deploy. Declarative vs. imperative programming: 4 key differences. `https://codefresh.io/learn/infrastructure-as-code/declarative-vs-imperative-programming-4-key-differences/`. Accessed 25-04-2024.

[38] De Meester, Ben and Dimou, Anastasia and Kleedorfer, Florian. *The Function Ontology*. May 2023. `https://fno.io/spec/`.

[39] RDFLib Team. rdflib 7.0.0 documentation. `https://rdflib.readthedocs.io/en/stable/`, 2023. Accessed: 2024-04-24.

[40] Ontotext. What is graphdb? `https://graphdb.ontotext.com/documentation/10.6/`, April 2024. Accessed: 2024-04-25.

[41] Apache Jena. Apache jena fuseki. `https://jena.apache.org/documentation/fuseki2/index.html`. Accessed 18-05-2024.

[42] Dylan Van Assche and Christophe Debruyne. BURPing through RML test cases. In *Submitted to Fifth International Workshop on Knowledge Graph Construction@ESWC2024*, 2024. under review.

[43] W3ID Community. Permanent identifiers for the web. `https://w3id.org/`. Accessed: 24-05-2024.

[44] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts*. McGraw-Hill, New York, 7 edition, 2010.

Appendix A

Structure of a Jupyter Notebook in JSON Format

Jupyter notebooks are stored in JSON format, and the structure of the JSON format can vary depending on the environment used to create and execute the notebook. In this work, we focus on notebooks created and run on Google Colab. The typical structure of those notebooks is illustrated in Listing A.1.

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {
     "id": "LSMxxEuDH2yt"
   },
   "source": [
     "## Hello world!"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 1,
   "metadata": {
    "colab": {
     "base_uri": "https://localhost:8080/"
    },
    "id": "qx-hpBoYPga0",
    "outputId": "fefad968-9647-498f-ce97-4be474378ac7"
   },
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
     "text": [
      "Accuracy: 0.9666666666666667\n"
     ]
    }
   ],
   "source": [
    "y_pred = knn.predict(X_test)\n",
    "\n",
    "accuracy = knn.score(X_test, y_test)\n",
    "print(f\"Accuracy: {accuracy}\")"
   ]
  }
 ],
 "metadata": {
  "colab": {
   "provenance": [],
   "toc_visible": true
  },
  "kernelspec": {
   "display_name": "TFE",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.12.0"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 0
}
```

Listing A.1: Structure of a Jupyter notebook in JSON format.

Appendix B

Joins

Relational algebra is a language that aims at performing operations, including joins. The join operation is used to associate several relationships. The `equi-join` is a conditional join that will perform the join operation based on an equality. Tables B.1 and B.2 show an example of the `equi-join` operation between two tables, where the equality condition is,

$$\texttt{T1.id\_Cell} = \texttt{T2.id\_Cell}$$

Table B.1: Example of two tables to join. On the left is the table `Notebook` (T1) and on the right, the table `Cell` (T2).

id_Notebook	id_Cell
ID N1	ID Ci

id_Cell	att1	att2
ID C1	att1 C1	att2 C1
ID C2	att1 C2	att2 C2
...	...	...
ID Ci	att1 Ci	att2 Ci
...	...	...
ID Cn	att1 Cn	att2 Cn

Table B.2: Result of the `equi-join` operation between the tables `Notebook` and `Cell`.

id Notebook	id Cell	att1	att2
ID N1	ID Ci	att1 Ci	att2 Ci

The `theta-join` is a conditional join that will perform the join operation based on a specified condition. Table B.3 shows the results of the `theta-join` operation between the two tables illustrated in Tables B.1 [44], where the condition is,

$$\texttt{T1.id\_Cell} = \texttt{T2.id\_Cell} \lor \texttt{T1.id\_Cell} \neq \texttt{T2.id\_Cell}$$

Table B.3: Result of the desired `theta-join` operation between the tables `Notebook` and `Cell`.

id Notebook	id Cell	att1	att2
ID N1	ID C1	att1 C1	att2 C1
ID N1	ID C2	att1 C2	att2 C2
ID N1	...	...	...
ID N1	ID Ci	att1 Ci	att2 Ci
ID N1	...	...	...
ID N1	ID Cn	att1 Cn	att2 Cn