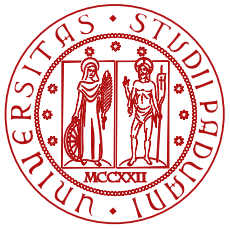


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

Analysis and evaluation of exploratory search workflows on Wikidata

Candidate

Gianmarco Prando

Student ID 2019170

Supervisor

Prof. Gianmaria Silvello

University of Padova

Co-supervisor

Prof. Matteo Lissandrini

Aalborg University

Academic Year
2021/2022
Graduation Date
05/12/2022

To my grandparents
and family

Abstract

With the evolution of the Web there are more and more people using it as a landmark to search for information and learn new things. Open Knowledge Bases, therefore, have become the cornerstone of everyday web use and consequently a very interesting subject to study and to introduce into degree courses in order to bring students closer to this topic. Database 2 course, held in the academic year 2021/2022, part of the Web Information and Data Engineering curricula of the Computer Engineering Master Degree, presented this topic to students for the first time. A large part of the course was dedicated to let students put their effort to investigate the world of the Semantic Web and the Knowledge Bases doing projects on them. In particular, one of the project, required to do exploratory search on Wikidata, one of the biggest Open Knowledge Base,through some workflows. This thesis gather the entire work did by the students and starting from this, create a model that allows a statistical analysis of the different processes that led the students to complete their project. Finally, create reference workflows to evaluate those made by students.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Snippets

List of Acronyms

CSV Comma-separated values

DBMS Database Management System

RDF Resource Description Framework

W3C World Wide Web Consortium

URI Uniform Resource Identifier

BGP Basic Graph Pattern

GGP Group Graph Pattern

SQL Structured Query Language

RDBMS Relational Database Management System

1 Introduction

Since its inception, the Web has been a useful tool. Today, its use is so heterogeneous that one can save simple files, stay up-to-date with current topics and even meet people. Years ago people used to learn word and things using their own printed dictionary, or through a local encyclopedia. Given that many dictionaries and encyclopedias move online becoming public knowledge, people started to use them as an everyday activity. Given the huge amount of data and the fact that being in the public domain several people can add and modify them, it was necessary to introduce new standards to handle those resources.

This new concept of open knowledge in the Web has become a field of study for many researchers of many Universities. Since this topic has been of interest for several years now, we must start educating students to bring them closer to this evolution of the way data is handled on the Web, teaching them new standards and technologies. During the last year, the Master Degree in Computer Engineering at the University of Padua changed, becoming more structured and providing different study paths. Curricula were introduced according to the macro subject covered throughout the degree course. One of these curricula is Web Information and Data Engineering, which focuses on web applications, search engines and databases. In particular there are two database courses: the first one covers the basics of tabular relational databases and relational algebra. The second instead, covers topics closer to the present day, such as the Semantic Web, Open Knowledge Bases and the new standards used to manage and share data on the Web. Given that the course is one of the last taught during the master degree, it is assumed that students, at this point

in their academic career, have a solid knowledge of programming and relational databases thus, to make students more enthusiastic about the subject, the idea was to decrease the theoretical load of the examination by favouring individual and group projects on the course topics. In particular, the individual project required studying one of the largest Open Knowledge Bases, Wikidata, only querying it, creating workflows on different topics. As a result, the work did by the students during their project, is the subject of this thesis.

Before entering the practical aspect of the thesis, I will present the background scenario, starting from graph databases, how data are represented in the Semantic Web and how our Knowledge Base of interest, Wikidata, is composed. I will also show the main concept and usage of SPARQL, the query language used to query graph databases. Moreover, I will present in detail the individual project that produced the data for this thesis, which were the assignments and how workflows are composed and distributed over students.

Then, I proceed showing my work. Starting from the data processing to collect the workflows information, I will discuss the structure used to handle all those information and then there are some statistics on that. In addition, I will show the creation of the reference workflow for each topic, in order to have a well-done and compact procedure of exploration. Finally, I will present the evaluation stage, which uses the reference workflows to evaluate other workflows on the same topic.

2

Background

The data accessible from the WEB grow every year and this leads to store them in the most suitable way, easy to manage and easy to query. Sometimes it is perfectly known what our database is going to store because we have well defined entities and relations, thus once the database schema has been defined, it will rarely be changed. Furthermore, the data are managed by a private enterprise, so that no person or application outside this specific mini-world can interact with them. Hence, the proper way to store the data is through a tabular relational database. This type of database also guarantees perfect consistency of stored data thanks to numerous constraints that prevent errors when inserting or updating data. If we try to store data that does not perfectly fulfil each constraint, the Database Management System (DBMS) responds with an error and with a simple rollback operation it is possible to return to the previous safe state. On the other hand, we can have situation where entities and relations can change over the time and also properties related to the entities can change due to a frequent evolution of the database schema. Relational database are not thought for frequent schema evolution: every time we have to drop a column or remove a relation between tables we must take care about all the constraints because data must be consistent and when we deal with hundreds of tables drop a simple column can waste a lot of hours. Furthermore, if we consider highly connected data, or even worst recursive relations (think about the friendship relation between people) there is a substantial grows of the time needed to query because we need to perform joins between tables and if we combine a lot of data with a lot of joins the result is a lot of time to query the database.

Moreover, if the domain of the data is no longer a mini-world but the data are public knowledge where each person and application can access and modify them, things become even worse with tabular relational databases. This is the case of Linked Open Data on the Semantic Web. The goal of the Semantic Web is to make Internet data machine-readable and a Web of data consists of data linked together so that they can be found, browsed, crawled and integrated. Linked Open Data are data available on the web with a machine-readable structure which use open standards to identify things and are linked to other people's data.

This new way of conceiving Web data lead to think to another way to store and represent data. When we have to deal with massive quantity and highly connected data the best way to represent them is through a graph, where the nodes are the objects and the edges are the relations between the objects. The information of the entities can be stored as properties of the nodes. One of the best examples of usage of the graph databases is a graph of friendships.

Let consider the example in Figure 2.1: the goal is to find all the friends of Jack. If we are using a graph database, we just need to find the node representing Jack and look at all outgoing arcs labeled *Friend*. The objects pointed by the arcs are all the friends of Jack.

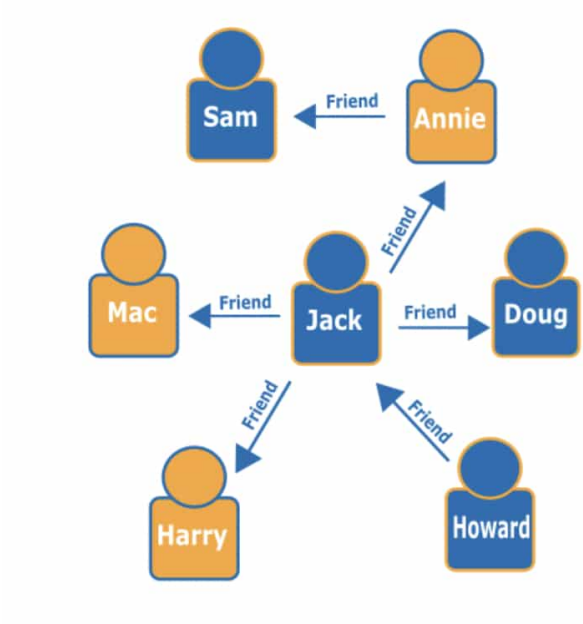


Figure 2.1: Friendship graph example

Of course graph databases are not the substitutes of the tabular relational

databases but they are a useful alternative with highly connected open data and object oriented data.

2.1 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a standard introduced by World Wide Web Consortium (W3C) [1] as a data model for representing interconnected data on the web. Linked Open Data are represented on the web using RDF statements (also called triples) hence, RDF is the standard for making statements about resources, which can be objects, documents, people or concepts. A collection of RDF statements about related entities can be used to construct an RDF graph that shows how those entities are related. Each triple consists of a subject, a predicate and an object and Figure 2.2 shows graphically how an RDF triple is composed.

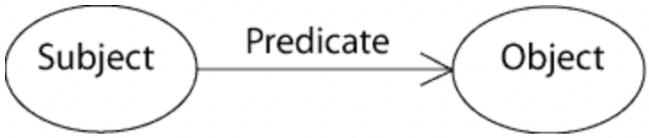


Figure 2.2: Triple example

The data on the Semantic Web are uniquely identified through the use of Uniform Resource Identifier (URI)s. Given that RDF is the standard to represents Linked Open Data on the Web, URIs are usually used in RDF statements. A URI is a unique sequence of characters that identifies a logical or physical resource used by web technologies. URIs may be used to identify anything, including real-world objects, concepts, or information resources. The subject of a triple may be an URI or a blank node, that represents an anonymous resource for which no URI or literal is provided. The predicate is an URI which also indicates a resource, representing a relationship between the subject and the object. The object may be an URI, blank node or a string literal. The literals can be of two type:

- **plain literal** is a string that may be combined with a language tag
- **typed literal** is a string combined with a datatype URI

In the simple graph example showed in Figure 2.1, it is possible to identify the set of nodes (people) that always play the role of subject or object and the set

of directed edges (the friend relation) that play the role of predicate. To describe this graph in RDF format we need to write an RDF triple for each relation. For example one triple of the graph is Annie (subject) - friend (predicate) - Sam (object).

RDF can be stored in several ways, but the most popular are:

- **Turtle** [2]: a compact, human-friendly format
- **N-Triples** [3]: a very simple, line-based format that is not as compact as Turtle
- **RDF/XML** [4]: an XML-based syntax that was the first standard format for serializing RDF

(a) URIs example (b) Curie URIs example

Figure 2.3: Graph examples

Even if RDF/XML was the first syntax introduce by W3C other RDF serializations are now preferred. Turtle allows an RDF graph to be written in a compact and natural text form, with abbreviations for common usage patterns and datatypes. In order to understand Turtle syntax, before entering the details, other components must be defined: Curie notation and namespaces. Curie notation is an extension mechanism based on the concept of scoping where names are created within a unique scope, and that scope's collection is managed by the group that defines it. It is comprised of two components, a prefix and a reference separated by the colon character. A namespace is a set of names that are used to identify and refer to objects of various kinds. They ensures that all of a given set of objects have unique names so that they can be easily identified. Namespaces and Curie notation are used to represent URIs in a compact manner.

Graphs in Figure 2.3 show both the ways of representing URIs. On the left (Figure 2.3a) each elements of the triples is represented with its complete URI. In the example, it is possible to identify two namespaces, that are `https://www.dbpedia.org/page/` and `https://schema.org/`. Hence, these namespaces can be used in a Curie notation defining the prefixes `dbpedia:` and `schema:`. The example on the right (Figure 2.3b) shows the same graph using the Curie representation of the URIs.

The graph in Figure 2.3b can be easily translated in Turtle syntax, given that Turtle uses the Curie notation. Each Turtle file begins with a preamble where prefixes are defined. Then, it proceeds triple by triple, separating them by the final period. Figure 2.4 shows the translation of the graph in Figure 2.3b.

```
@prefix schema: <https://schema.org/>.
@prefix dbpedia: <https://dbpedia.org/page/>.

dbpedia:Emma_Stone schema:actor dbpedia:IrrationalMan .
dbpedia:Woody_Allen schema:director dbpedia:IrrationalMan .
dbpedia:Woody_Allen schema:director dbpedia:AnnieHall .
dbpedia:Woody_Allen schema:actor dbpedia:Manhattan .
dbpedia:Woody_Allen schema:director dbpedia:Manhattan .
```

Figure 2.4: Turtle serialization example

2.2 Exploratory Search

Exploratory search is a specialization of information exploration which represents the activities carried out by searchers who are: [5]

- unfamiliar with the domain of their goal (i.e. need to learn about the topic in order to understand how to achieve their goal) or
- unsure about the ways to achieve their goals (either the technology or the process) or
- unsure about their goals

Gary Marchionini, in the paper "Exploratory search: from finding to understanding" of the 2006 [6], depicts the activity of exploratory search as the union of two other activities: **learn** that includes knowledge acquisition, comprehension/interpretation, comparison, aggregation/integration and **investigate** that

includes accretion, analysis, exclusion/negation, synthesis, evaluation, discovery, planning/forecasting and transformation. He also wrote that, since a lot of materials are available online, search to learn became very important. Learning searches are activities that usually require several iterations and return objects that must be understand, interpreted and processed. These objects can be heterogeneous and the user needs to spend time to compare and evaluate the information that had been retrieved. These tasks that user performs, define what exploratory search is: look for information, comprehend it, analyzed and evaluate the information. After this cycle of operation, the knowledge of the user increase.

Exploratory search over graph databases is in fact a way to investigate and learn. Find information in a graph requires many iterations because users usually start from a node of the graph to explore and investigate the graph and they cannot expect that what they are looking for is immediately connected to the starting node. In fact, for each iteration of the search, after doing the query, users must understand the result set of the answer, analyze all the objects in the sets and try to plan the next query according to their evaluation and intuition. This is the aim of the exploratory search: acquire knowledge, comprehend concepts, interpret ideas and compare and aggregate data.

Knowledge graphs are the best way to learn through exploratory search. A knowledge graph is *a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent potentially different relations between these entities*. Open knowledge graphs are published online, making their content accessible for the public good. Some of the best known are DBpedia [7] and Wikidata [8].

2.3 Wikidata

One of the biggest knowledge graph is Wikidata, hosted by the Wikimedia Foundation. Wikidata is an open knowledge base and everyone can interact with it.

2.3.1 History

The Wikimedia Foundation, Inc. is an American nonprofit organization headquartered in San Francisco, California [9]. The Wikimedia Foundation was

established in 2003 in St. Petersburg, Florida, by Jimmy Wales as a nonprofit way to fund Wikipedia. Their mission is to "empower and engage people around the world to collect and develop educational content under a free license or in the public domain, and to disseminate it effectively and globally." [10]

Wikidata was launched on 29 October 2012 and at this time the items that could be created were very basic (an example of the page layout of this initial phase is showed in Figure 2.5). The information of an item were:

- label, a name or title for the item
- aliases, alternative terms for the label of the item
- description
- links to articles about the topic in all the various language (also called interwiki links)

Figure 2.5: Example layout at the beginning

On 4 February 2013, statements were introduced to Wikidata entries. This was a big upgrade because until this time the items inside Wikidata did not contain a lot of information. Properties on items helped to enrich them with more useful content. The possible values for properties were initially limited to two data types (items and images on Wikimedia Commons), with more data types (such as coordinates and dates) to follow later.

On 7 September 2015, the Wikimedia Foundation announced the release of the Wikidata Query Service, which lets users run queries on the data contained in Wikidata.[11] The service uses SPARQL as the query language.

The Foundation finances itself mainly through millions of small donations from Wikipedia readers, collected through email campaigns and annual fundraising banners placed on Wikipedia.

2.3.2 Item's Structure

Wikidata is a document-oriented database where each document is an item which represent any kind of topic, concept or object.

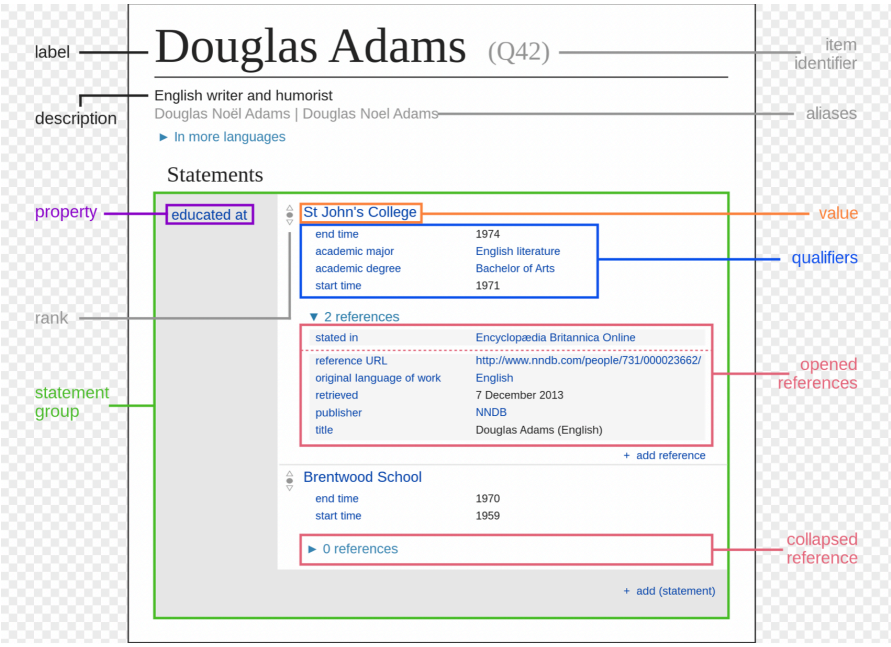


Figure 2.6: Wikidata Item's Structure

Figure 2.6 describes how an item is structured in Wikidata. It is possible to distinguish two main parts:

- the first part is mandatory and consists of a unique positive integer number prefixed by the upper-case letter Q (the QID) related to a label and a description. The description briefly describes the item and it is often written in more than one language
- the second part is optional. There can be some aliases that are synonyms of the label or other way to recognize the label and a list of statements which are information known about the item

The label of the items does not need to be unique. One possible example is a singer that produced an album with his own name. In this case the singer and the album are two different items with the same label. The crucial thing is

that the description must be different if the label is the same, because the QID is linked to the combination of label and description.

The statements are a list of key-value pairs which match a property with one or more entity values. In the example of Figure 2.6, the statement

Douglas Adams was educated at St John's College 1

is translated in Wikidata with a statement composed of the property *educated at* (key) and the object *St John's College* (value) of the item *Douglas Adams*. Properties in Wikidata are identified by a positive integer number prefixed by the capital letter P (PID). As the Wikidata items, each property has its own Wikidata page with optional label, description, aliases and statements. Properties may define some rules, namely the data type of the value to avoid basic inconsistency (e.g. the value of the property date of birth must be a date) or single value constraint, since there exist properties that typically have only one value (e.g. the continent of a country). On the other hand, there exist properties with more than one value, for example a company can have more than one founder or in the previous statement example (1) *Douglas Adams* was educated at two different colleges/schools. In these cases, qualifiers play a very important role. They are optional, but when there are several values associated to the same property, qualifiers provide more information and meaning to the value. The qualifiers *start time* and *end time* in Figure 2.6 associated to the value *St John's College* provide useful temporal information, while the *academic degree* provide the type of education *Douglas Adams* did in that college.

Values can be other Wikidata objects - *St John's College* in the example just made refers to another Wikidata item - or other values such as integers, strings or dates.

Finally values in the statements may be annotated with some references, that are resources which contains the information of the statement's content.

2.3.3 Wikidata RDF Graph

The structure of the Wikidata items described in the Subsection 2.3.2 can be seen as an RDF graph given that each statements of each item can be translated into an RDF triple with these simple operations:

- the Wikidata items become the subject of the triple

- the properties of the statement become the predicate of the triple
- the values of the properties become the object of the triple

Items and properties are uniquely identified in Wikidata, i.e. they have an associated URI. There are two standards:

- items' URI standard is

  https://www.wikidata.org/entity/{item-identifier}

  where the item-identifier is replaced with the QID of the item. Douglas Adams' URI is `https://www.wikidata.org/entity/Q42`

- properties' URI standard is

  https://www.wikidata.org/prop/direct/{prop-identifier}

  where prop-indentifier is the PID of the property. The URI of the property `educated at` is `https://www.wikidata.org/prop/direct/P69`

The object of the triples in RDF can be an URI, that are Wikidata items, or a literal value that are, as said above, integers, strings or dates.

The first part of the URIs is the namespace (`https://www.wikidata.org/entity/` for the items URI) and it is common between all the Wikidata items.

3 Technologies

3.1 SPARQL

SPARQL Protocol and RDF Query Language, or simply SPARQL, is a query language used to retrieve and manipulate data stored in RDF format. It is a standard introduced by W3C in 2008 and successively updated in 2013 [12] and it is one of the main technologies of the Semantic Web.

A SPARQL query is composed of:

- optional prefixes
- selection
- pattern matching
- optional aggregation functions, ordering function or limits to the result set

3.1.1 Prefixes

Since entities and predicates in RDF are identified by URIs, namespaces are used to identify the first part of the URI and to not write the entire namespace every time SPARQL uses prefixes that relate a short string to a namespace just like in the Turtle files so that each URI can be expressed using Curie notation. Considering the statement of the Subsection 2.3.2 (1), Douglas Adams was educated at St John's College, there are three URIs:

- Douglas Adams is `http://www.wikidata.org/entity/Q42`
- educated at is `http://www.wikidata.org/prop/direct/P69`
- St John's College is `http://www.wikidata.org/entity/Q691283`

It can be possible to define two different prefixes:

- **wd:** referring to the namespace `http://www.wikidata.org/entity/` and it is meant for items
- **wdt:** referring to the namespace `http://www.wikidata.org/prop/direct/` that is used for properties

Using these prefixes the triple can be written as *wd:Q42 wdt:P69 wd:Q691283*.

3.1.2 SPARQL Constructs

There are several constructs that can be used in SPARQL queries devoted to different purposes. The keyword `SELECT` usually anticipates a list of variables that are served in the result set. Selection can be used with the keyword `DISTINCT` if the scope of the query is not to retrieve the same result multiple times. A variable of the result set can also be an aggregation function, such as `COUNT`, `SUM` or `GROUP_CONCAT`.

`ASK` provides only two type of answer, yes or no (or alternatively true/false). It returns yes (true) if the pattern exists. From the statement translated in the previous subsection if it were run on Wikidata an ask query that try to match the pattern *wd:Q42 wdt:P69 wd:Q42* the result will return no (false) because this triple does not exists in Wikidata.

`DESCRIBE` returns a single result RDF graph containing all RDF data about resources. The `DESCRIBE` form takes each of the resources identified in a solution, together with any resources directly named by IRI, and assembles a single RDF graph by taking a description which can come from any information available including the target RDF Dataset. The description is determined by the query service.

The `CONSTRUCT` query form returns a single RDF graph specified by a graph template. The result is an RDF graph formed by taking each query solution in the solution sequence, substituting for the variables in the graph template, and combining the triples into a single RDF graph by set union.

3.1.3 Pattern Matching

The keyword `WHERE` anticipates a block of triple patterns. Triples inside this block are in the same form of RDF triples but all the three elements can be a variable. SPARQL is based on graph pattern matching and the simplest is the Basic Graph Pattern (BGP). A BGP is a set composed by one or more triple patterns. More BGPs define a Group Graph Pattern (GGP). Elements in triple patterns can be URI, literal values or variables. A variable must be prefixed with the question mark and it can be used one or more time inside the graph matching. Once a variable is used its value is bound and if the same variable come afterwards the value bound before will be used. It is important to remark that SPARQL query engine performs a conjunction between triple patterns. The code snippet 3.1 shows a SPARQL query example.

```
1 PREFIX wd:<http://www.wikidata.org/entity/>
2 PREFIX wdt:<http://www.wikidata.org/prop/direct/>
3
4 SELECT DISTINCT ?country
5 WHERE{
6     ?people wdt:P69 wd:Q691283.
7     ?people wdt:P27 ?country.
8 }
```

Code 3.1: SPARQL query example

In the very beginning of the query there are the two declarations of the prefixes, as explained in the Subsection 3.1.1. After them there is the `SELECT` keyword followed by `DISTINCT` and one variable: thus it is clear that after the graph pattern matching the query will return all the distinct elements inside the set bound in the variable *country*. The BGP to be matched is composed of two triple patterns. The subject of the first pattern match is a variable, and after this first matching it will contain all the subjects of triples which have as predicate *wdt:P69* and *wd:Q691283* as object. The second pattern has a new variable *country* as object and uses the variable *people* just bound as subject. After the match the variable *country* will contain all the objects of triples which have as subject one of the element belonging to the set of the *people* variable and *wdt:P27* as predicate (that is the country of citizenship in Wikidata). More complex queries are possible, and BGP and GGP can be used also as subquery.

After the pattern matching it is possible to group variables and use aggregation functions on them, order the results according to both ascending or

descending order and limit the result to maximum number of elements in the result.

3.2 Virtuoso Universal Server

OpenLink Virtuoso [13] is a cross platform Universal Server that implements Web, File, and Database server functionality as a single server solution. It includes support for almost all the main standards of the Web and Data Access such as XML, XPATH, ODBC and many others. Virtuoso currently supports all the main Operating systems, that are Windows, Linux and MacOS. Virtuoso is very powerful because with a single connection it is able to connect simultaneously different client applications to many different databases and treats them as it were a single database. Even if Virtuoso supports RDF it is important to remark that it is based on a Relational Database Management System (RDBMS) [14] with a column store architecture [15]. This imply that columns of a table are stored contiguously, hence values of the same column on consecutive rows are physically adjacent. Since the graph database is stored in a relational database it is important to define some indexes to speed up the access query time. The index scheme consists in five indexes, two of them are full indexes while the other three are partial indexes. Virtuoso favours query where the predicate is specified in the triple pattern and if the object or the subject are also specified, the retrieve is very efficiently.

The choice of Virtuoso is also confirmed by a recent paper of the Wikidata Query Service Search Team [16] because this year they decided to replace the current Blazegraph backend to a new more efficient one. They analyze more than 20 different backends and they judged them with a score on different criteria. In the paper they report only the best four and Virtuoso was one of the four with very high scores on scalability.

4

Project and Data

Data for this thesis came from the Database 2 course held in the academic year 2021/2022 which is part of the Web Information and Data Engineering curricula of the Computer Engineering Master Degree. The course covered different topics of the Semantic Web, such as Linked Open Data, RDF, graph databases, SPARQL and exploratory search. The final grade consisted of the written examination, two group projects and an individual project. My thesis investigates the work carried out by the students during their individual project.

4.1 Project Assignment

The aim of the project was to let the students to practice with SPARQL and the idea was to have them do exploratory search on Wikidata. In order to do this, professors Gianmaria Silvello and Matteo Lissandrini, prepared some workflows on different macro-topics. The macro-topics were:

- sport
- movies
- geography
- companies
- books
- politics

Each students had 6 different workflows to complete, one per macro-topic. The work environment was the Jupyter Lab Web Interface installed in a remote virtual machine. The workflows were actually python notebooks which had a specific assignment according to the different macro-topic and the goals of that workflow. Each workflow has an initial common part which explained the general instructions about exploratory search on Wikidata and what students were allowed to do inside the python notebook. Furthermore, there was an initial python cell which contained the python imports needed, a string named *prefixString* and the two functions to run the queries, the first one used for generic queries and the other one used only for the ask query. The functions are reported in the code snippet 4.1.

```
from SPARQLWrapper import SPARQLWrapper, JSON

prefixString = """
##-cda4aeaa8e-##
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX sc: <http://schema.org/>
"""

# select and construct queries
def run_query(queryString):
    to_run = prefixString + "\n" + queryString
    sparql = SPARQLWrapper("http://a256-gc1-02.srv.aau.dk:5820/sparql
    ")
    sparql.setTimeout(300)
    sparql.setReturnFormat(JSON)
    sparql.setQuery(to_run)

    try :
        results = sparql.query()
        json_results = results.convert()
        if len(json_results['results']['bindings'])==0:
            print("Empty")
            return 0
        for bindings in json_results['results']['bindings']:
            print( [ (var, value['value'])  for var, value in
    bindings.items() ] )

        return len(json_results['results']['bindings'])

```

```
29      except Exception as e :
30          print("The operation failed", e)
31
32  # Ask queries
33  def run_ask_query(queryString):
34      to_run = prefixString + "\n" + queryString
35
36      sparql = SPARQLWrapper("http://a256-gc1-02.srv.aau.dk:5820/sparql
        ")
37      sparql.setTimeout(300)
38      sparql.setReturnFormat(JSON)
39      sparql.setQuery(to_run)
40
41      try :
42          return sparql.query().convert()
43      except Exception as e :
44          print("The operation failed", e)
```

Code 4.1: Python functions to run the SPARQL queries

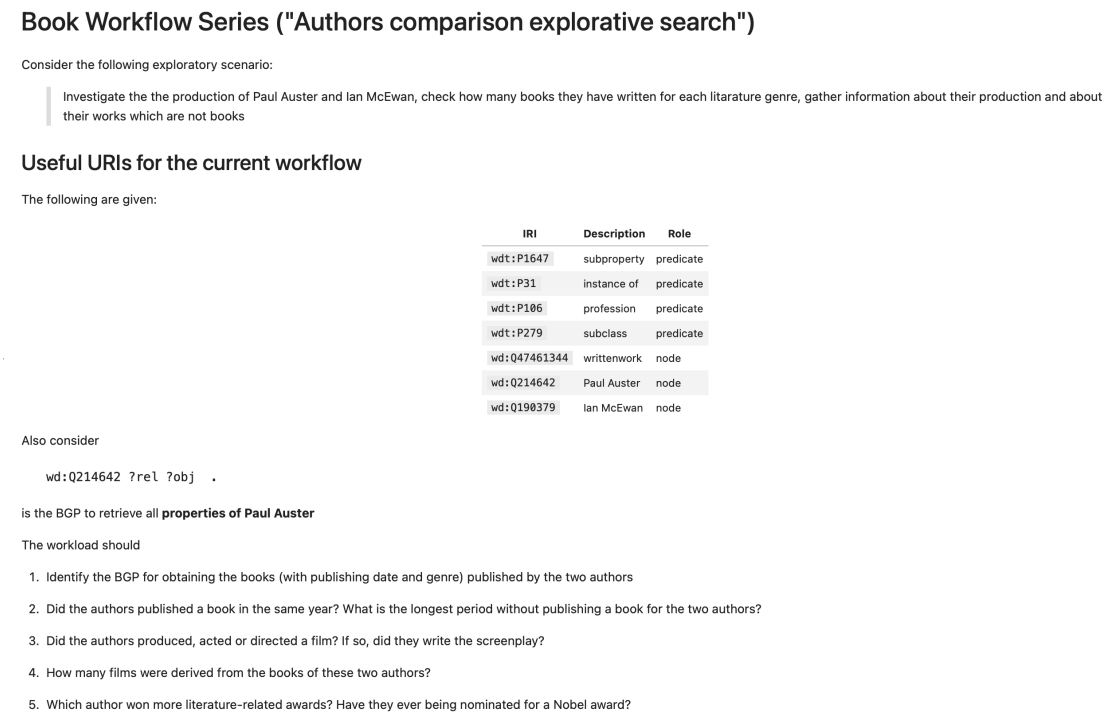
The *prefixString* defined two important things:

- **prefixes**: as explained in the Subsection 3.1.1 the SPARQL query can starts with some prefixes definition to use later in the query text. Here are defined all the prefixes needed that are one for the Wikidata entities and another one for the properties. There is also another last prefix which is mainly used to get a human-readable name of a property or a entity in Wikidata. The benefit to have the prefixes here is that students did not care about prefixes and thus they must concentrated only on the query text
- **notebook code**: in the very beginning, before the prefixes definition (line 4 of the code snippet 4.1), there was the python notebook filename, delimited by two dashes. Practically the students, before to start writing queries, manually put the notebook filename in this area

The function *run_query* takes one parameter *queryString*, that is the query text. It defines the string to run in the database as a concatenation of the *prefixString* and the *queryString*. Then it creates a SPARQLWrapper object with the URL to the SPARQL endpoint and it sets a timeout for the query execution, the format of the response and the string to run as SPARQL query previously defined. Finally it try to get the response if there are no syntactic errors in the query text or other type of errors such as timeout expiration, and it prints the results. If there are any kind of errors it catches the exception and it prints an error message that describes the error. The *run_ask_query* basically do the same, the only difference

is the interpretation of the result, since an ask query reply only with True or False.

After this python cell there is a markdown cell which briefly describes the macro-topic (e.g. movies) of the workflow and the specific topic of that workflow, with some indications about what are the exploratory information needed. Then there is a table containing some useful URIs given for the current workflow. Some of those URIs are widely used and they were given to all the workflows (one of the most important is the predicate *instance of*), some others are specific for the workflow and they were usually the starting point of the exploration. Finally, there are some questions that the students had to answer doing exploratory search. Figure 4.1 shows an example of a workflow assignment on the macro-topic Book.



Book Workflow Series ("Authors comparison explorative search")

Consider the following exploratory scenario:

> Investigate the the production of Paul Auster and Ian McEwan, check how many books they have written for each litarature genre, gather information about their production and about their works which are not books

Useful URIs for the current workflow

The following are given:

IRI	Description	Role
wdt:P1647	subproperty	predicate
wdt:P31	instance of	predicate
wdt:P106	profession	predicate
wdt:P279	subclass	predicate
wd:Q47461344	writtenwork	node
wd:Q214642	Paul Auster	node
wd:Q190379	Ian McEwan	node

Also consider

```
wd:Q214642 ?rel ?obj  .
```

is the BGP to retrieve all **properties of Paul Auster**

The workload should

1. Identify the BGP for obtaining the books (with publishing date and genre) published by the two authors
2. Did the authors published a book in the same year? What is the longest period without publishing a book for the two authors?
3. Did the authors produced, acted or directed a film? If so, did they write the screenplay?
4. How many films were derived from the books of these two authors?
5. Which author won more literature-related awards? Have they ever being nominated for a Nobel award?

Figure 4.1: Example of the python cell which contains the title and the assignment for the workflow

The are usually five to ten tasks for each workflow, depending on the workload of each task. Some tasks required a big exploration of the graph, while others were simpler. In general the global workload of each workflow was well balanced.

4.2 Available Data

The individual project in the academic year 2021/2022 was assigned to 24 students but 3 of them did not even start the project. Professors prepared 24 different workflows totally, 4 for each macro-topic. Hence, there is a redundancy because the same workflow has been done by 4 to 6 different students. The total number of completed workflows submitted by the deadline is 126, and the Table 4.1 shows how they are divided by macro-topic and topic.

One other important available data was the query log, that contains all the real queries ran by the students. It stored the complete text of the query sent to virtuoso, including prefixes and the hexadecimal code to identify the origin of the query. To split the dimension of the documents containing the logs, the system created one log file per day.

Geography Macro Topic	
Topic name	**# workflows**
Archaeological Sites	6
European Cathedrals	4
American Architects	5
Place of Birth, Death, and Burial	6
Sport Macro Topic	
Topic name	**# workflows**
F1 pilots	6
World Records	4
Olympic	6
FIFA World Cup events	5
Politics Macro Topic	
Topic name	**# workflows**
Presidents of countries	6
International Treaties	4
Politicians in E.U.	6
Monarchies	5
Book Macro Topic	
Topic name	**# workflows**
Nobel laureates	6
Political Magazines	4
Authors Comparison	6
Author Comparison	5
Companies Macro Topic	
Topic name	**# workflows**
IT Companies	6
Business People in Germany	4
Economy of EU States	6
Trademarks across the world	5
Movie Macro Topic	
Topic name	**# workflows**
Directors	6
The Batman movies	4
Horror Franchises	6
Tv series	5

Table 4.1: Available workflows for each topic

5 Data Processing

An important part of the thesis is dedicated to analyze and do statistics on the workflows, to understand how students thought and which were the topics that required more exploration to met the goal of the workflow. In order to do the final analysis and statistics it is possible to divide the work in some steps, which are:

- collect the workflows path
- collect the information in the workflows
- create a structure to represent the information
- keywords analysis

5.1 Path collection

Originally workflows were stored in two different server to lighten the load on them, thus in the very beginning I merged everything in one server. Each student had a folder named with the following convention: capital letter M followed by the student ID number. Each folder contained the python notebook files named with an hexadecimal string of 10 characters. The name of every notebook file is the $MD5$ hash function of the string obtained by the concatenation of the student ID number and the name of the workflow's topic. In the end there was a *notebook* folder that contained all the student folders (an example of the directory structure is showed in Figure 5.1).

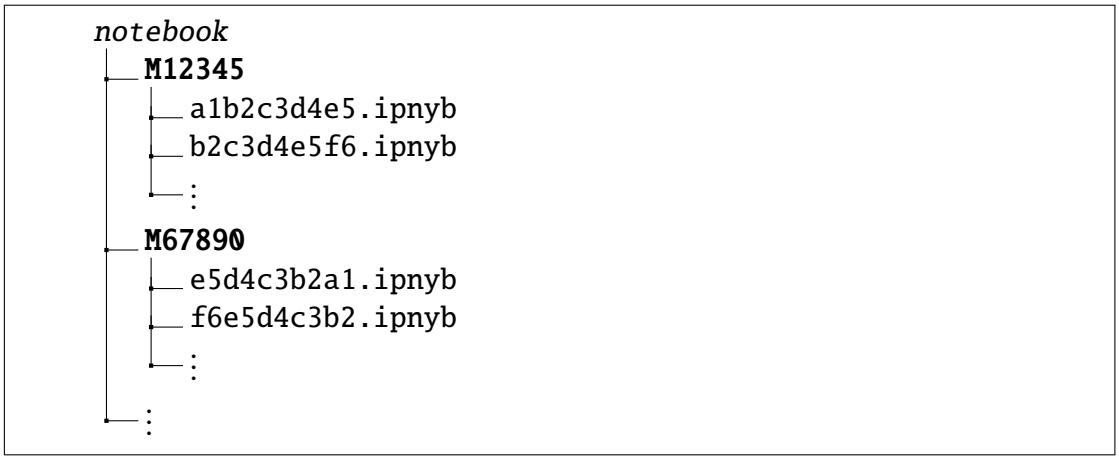


Figure 5.1: Directory tree example of how the workflows has been stored

In order to collect the file paths there is a function that explores the directory tree and in the end returns a dictionary which keys are the student IDs and the values are the lists of the file paths of the python notebooks did by the student.

5.2 Information Collection

The information collection of each workflow consists in three parts:

- read the title and translate the student IDs dictionary to a new topic based dictionary
- workflow tasks collection
- query collection

5.2.1 Dictionary Translation

The procedure explained in the Section 5.1 groups the file paths by student IDs but this exploration project operates with topic and macro-topic thus it is necessary to group all the python notebooks by their topic and also group each topic by their macro-topic. To produce an algorithm that was as generic as possible, the idea was to programmatically open each python notebook and read the cell that contains the title of the workflow, which is a concatenation between the macro-topic and the topic (an example can be seen in the first line of the Figure 4.1).

This was possible because the structure of the first part of every workflow was the same (as it is explained in the Section 4.1) and also because a python notebook is a JSON file that contains some metadata and an array representing the notebook cells. Each cell is represented as an array which elements are the rows of the cell. In order to extract the workflow's title it is sufficient to read the first line of the third python cell. The macro-topic is everything before the first parenthesis while the topic is what is written inside the parenthesis. At this point the procedure to translate the people based dictionary of the file paths to a new topic based dictionary is very simple and it is shown in the Algorithm 1. The algorithm returns a dictionary which keys the macro-topics: the value for each key is another dictionary which keys are the topics and the values are the file paths.

Algorithm 1 Student to Topic Algorithm

Require: $peopleDir$ the people based dictionary
Require: $getTitleFromNotebook(filePath)$ a function that given a file path of a python notebook return the title
Require: $getMacroTopicFromTitle(title)$ a function that given the workflows title return the macro topic
Require: $getTopicFromTitle(title)$ a function that given the workflows title return the topic
$workflows \leftarrow \{\}$
for $list$ in $peopleDir.keys()$ **do**
  for $filePath$ in $peopleDir[list]$ **do**
    $title \leftarrow getTitleFromNotebook(filePath)$
    $macroTopic \leftarrow getMacroTopicFromTitle(title)$
    $topic \leftarrow getTopicFromTitle(title)$
    if $macro - topic$ not in $workflows.keys()$ **then**
      $workflows[macroTopic] \leftarrow \{\}$
    end if
    if $topic$ not in $workflows[macroTopic].keys()$ **then**
      $workflows[macroTopic][topic] \leftarrow []$
    end if
    $workflows[macroTopic][topic]$.append($filePath$)
  end for
end for
return $workflows$

5.2.2 Tasks Collection

Each workflow contains a list of tasks which the student should answer. It is possible to see an example in the bottom part of the Figure 4.1. As for the title extraction also this list of tasks can be extract from the python notebook due to the standardization of this assignment part of the workflow. In particular the list is always introduced by the sentence "*The workload should*", so the algorithm that extract this list iterates amongst the rows until it will find the row which contains "*The workload should*". The structure of the *task* row is always a number then the dot and finally the text of the task, thus it is possible to recognize each task's text with the related number that in the end produce a dictionary which keys the number of the task and the value is the text of the task.

5.2.3 Query Collection

This is the most sophisticated part of the information collection because on the contrary of the assignment part of the workflow, this one has been written by students. Anyway students followed a standard when they wrote the notebooks and they always put one query for each python cell to avoid confusion. This is intuitive because in general there is the needed to run only one query at a time when doing Exploratory Search.

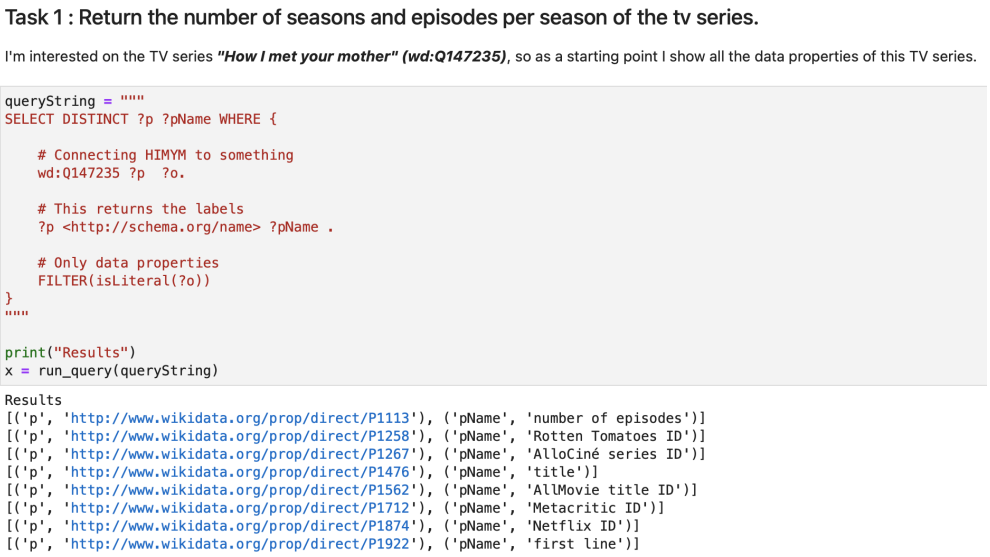


Figure 5.2: Example of a python cell which contains a query

The Figure 5.2 depicts an example of a cell which contains a query. Students

used to put a markdown cell before to explain what they are going to search with the following query and if the query was the first of the current task, they often put this information emphasizing the fact that they are starting with that specific task. Immediately after the text of the query, there is the output provided by the function *run_query* (explained in the Section 4.1).

In order to extract the queries from the notebook the procedure to open it is the same explained in the Subsection 5.2.1. At this point iterating through the code cells it is possible to find the queries, that must be cleaned and stored. Before to enter the details of the extraction it is important to clarify that an assumption was made: each student wrote the SPARQL queries assigning directly the string to a variable. They did not perform any concatenation and the string was usually written in several lines. The function to get the query from the python cell works as follows:

1. find the line with the command *run_query*
2. obtain the name of the parameter of the *run_query* function
3. go back in the iteration of the cell until a variable that match the name of the parameter is found.
4. from the line where the variable is assigned remove the name of the variable and the symbol "**=**"
5. verify if the string is in a single line or in multiple lines (in python a single line string starts and ends with one " character, while a multiple lines string starts and ends with three ", so it is sufficient to check this condition to understand the situation) and:
    - keep the whole line if there is a single line string, removing the initial and final " (very uncommon situation) or:
    - scan the next lines and concatenate them until the triple " is reached meaning that the string is finished

Then to distinguish between different tasks it is necessary to have a look at the markdown cells and check whether the cells starts with the keyword **Task** followed by a number. This cell announces the beginning of a new task. Overall, the query extraction is explained by the 5.1.

```
# return a dictionary of queries
def query_extractor(notebook,goals):
    #open the notebook
    data = json.load(open(notebook))
```

```
    #get the list of the cells
    df = pd.DataFrame(data['cells'])

    #skip the assignment cells of the notebook
    count = False

    #contains a list of lines of code. When the run_query command is found, go back in that list and find the related query
    container = []
    #the actual goal that student is analyzing
    actual_goal = ""
    #the dictionary of the queries
    query_dict = {}

    for row in df.itertuples():
        if row.cell_type == "markdown":
            for it in row.source:
                ## verify if we can start to consider the cells
                if "useful uri" in str(it).lower():
                    count = True
                elif count:
                    #looking for a markdown cell that tells me which task people are doing
                    if "task" in str(it).lower():
                        task_line = str(it).lower()[(str(it).lower().index("task")+5):]
                        for g in goals:
                            if task_line.startswith(g):
                                #task found
                                actual_goal = g
        elif row.cell_type == "code":
            # code cells. Looking for the queries
            for it in row.source:
                if not count:
                    continue
                if "run_query" in str(it).lower():
                    ##run query command --> extract the query
                    param = get_function_parameter(it)
                    query_str = get_query_text(container,param)
                    #append the string in the list of queries of the related goal
                    if actual_goal not in query_dict:
                        query_dict[actual_goal] = [query_str]
```

```
                    else:
                        query_dict[actual_goal].append(query_str)
                    container = []
                elif "run_ask_query" in str(it).lower():
                    ##run ask query command --> extract the query
                    param = get_function_parameter(it,True)
                    query_str = get_query_text(container,param)
                    #append the string in the list of queries of the
    related goal
                    if actual_goal not in query_dict:
                        query_dict[actual_goal] = [query_str]
                    else:
                        query_dict[actual_goal].append(query_str)
                    container = []
                else:
                    container.append(it)
    ## at the end return the object
    return query_dict
```

Code 5.1: Python functions to extract the queries from the notebooks

Basically this function open the notebook and skip the first few cells with the information of the workflow and the assignments. Then it iterates through the cells and if the type of the cell is *markdown* it try to check if it is present the keyword **task**: if it is, it gets the number of the task, reading it immediately after the word **task** and through an auxiliary variable it keep trace of this number to use it later as key to store the queries. If the type of the cell is code, it means that probably there is a query inside and if the keyword *run_query* or *run_ask_query* is found, the function gets the text of the query passing an array of lines and the name of the parameter. Then it uploads the dictionary of the query appending the new query to a list stored with key the number of the task. Finally the function return a dictionary with keys the number of the tasks and values a list of queries associated to the task.

5.2.4 Query Keywords Analysis

An important part of this work was to fulfill analysis on the SPARQL keywords usage. The keywords can be grouped in different sets depending on their meaning and the once analyzed are:

- **Type of query**: select, ask, describe, construct, nested query

- **Result set restriction/order**: distinct, limit, offset, order by
- **Filters and set operations**: filter, and, union, optional, graph, exists, not exists, minus
- **Aggregation operations**: count, max, min, avg, sum, group by, having, group_concat

In order to check the presence of the keywords in a query there is a function that parse the text of the query and try to match each keyword: in the end the function return a bitmap. The ones in the bitmap correspond to presence of that keyword in the query. The bitmap's order is decided in the very beginning and it never change to guarantee consistency.

5.3 Representation of the Information

At this point all the functions to collect and extract the information from the notebook were designed and it is necessary to store all the information about the bitmaps in one single structure. The structure that represents everything can be viewed as a 6-dimensional array with the following meaning:

- the first dimension represents the macro-topic of the workflows, for example Sport or Movie
- the second dimension represents the sub topic of the workflows, for example the Director Exploratory Search on the movie macro-topic
- the third dimension represents the students
- the fourth dimension represents the goal/task of the topic
- the fifth dimension represents the query
- finally, the sixth dimension is the bitmap of the keywords usage

The shape of this solid is (6, 4, 21, 12, 138, 26), meaning that there are six macro-topic, at most four sub topic for each macro-topic, at most 21 students, at most 12 goals/tasks for each workflow, at most 138 queries for each workflow and 26 elements that are the keywords bitmap. The structure used to store the solid is a 6-dimensional NumPy [17] array. Before populating the solid, three maps are needed to relate indexes of the solid with macro-topic, sub-topic and students. To identify students and macro-topics a single index is needed, while to identify the sub-topic a pair of indexes is necessary because

a sub-topic uniquely depends from its macro-topic. These maps are stored in Comma-separated values (CSV) files which are useful later in the Chapter 6.

In order to use this big solid some functions are needed. The functions cover the following objects:

- **MACRO-TOPIC (1st dimension)**: there are two functions, the first one given the set of workflows and the specific macro-topic name, it converts it in a integer index to use in the solid. The second one does the opposite, given the set of workflows and the index it returns the related macro-topic's name
- **SUB-TOPIC (2nd dimension)**: there are three functions, the first one given the set of workflows and the specific name of the workflow topic, it converts it in a integer index to use in the solid. The second one does the opposite, given the set of workflows and the topic's index it returns the related topic's name. The last function given the topic's name it return its macro-topic
- **STUDENT (3rd dimension)**: there are two functions, the first one given the set of workflows, the specific topic name or the related topic index in the solid and the student ID, it converts it in a integer index to use in the solid. The second one does the opposite, given the set of workflows, the topic index or the topic name and the student's index it returns the related student's ID
- **GOAL (4th dimension)**: there are two functions, the first one given the set of goals and the number of the goal, it converts it in a integer index to use in the solid. The second one does the opposite, given the set of goals and the index it returns the related goal's number
- **QUERY (5th dimension)**: the function is thought to return the textual query given the workflow index, the student index, the goal index and the index of the query in the solid

The population of the solid is shown in the Algorithm 2 and it works by iteration through the dimension of the solid. It starts from the macro-topic and the topic and with the utility functions it retrieves both the indexes given the macro-topic's name and the topic's name. In addition, it computes the goals of the current topic before to iterate through the students. The students iteration starts getting the corresponding index of the student and the dictionary with all the queries text where the keys correspond to the goals of the topic. At this stage for each key (goal) in the dictionary it gets the index and it scans the related list of queries keeping a progressive index, that starts from zero, to use it as index for the query in the solid. Lastly, it creates the bitmaps of the SPARQL keywords and it adds it to the solid.

Algorithm 2 Solid Population Algorithm

Require: $workflows$ dictionary, returned by the Algorithm 1
Require: $getIndexByMacroTopic(macro)$ function to convert the macro-topic string into the index
Require: $getIndexByTopic(topic)$ function to convert the topic's string to the index
Require: $getIndexByStudent(student)$ function to convert the student's ID to the index
Require: $getStudentId(filepath)$ function to retrieve the student's ID from the filepath of the notebook's workflow
Require: $findWorkflowsGoal(nb)$ function that given a workflow's notebook return its goals
Require: $getIndexByGoal(goals, d)$ function to convert the goal's number to the index
Require: $analyzeQuery(q)$ function that check the SPARQL keywords in the query's text and return the correspondent bitmap

```
solid ← createSolid()
for macro in workflows.keys() do
    for topic in workflows[macro].keys() do
        indexMacro ← getIndexByMacroTopic(macro)
        indexTopic ← getIndexBySubTopic(topic)
        goals ← findWorkflowGoals(workflows[macro][topic][0])
        for nb in workflows[macro][topic] do
            indexStud ← getIndexByStudent(getStudentId(nb))
            queries ← queryExtractor(nb, goals)
            for d in queries do
                indexGoal ← getIndexByGoal(goals, d)
                index ← 0
                for q in dictionary[d] do
                    solid[indexMacro, indexWork, indexStud, indexGoal, index] ←
                    analyzeQuery(q)
                    index ← index + 1
                end for
            end for
        end for
    end for
end for
return solid
```

5.4 Keywords Analysis

As a result of this population function, it is possible - accessing the solid with the right indexes of the macro-topic, topic, student, goal and query - to obtain the bitmap with the information about the keywords used in that specific query. To analyze the solid and run statistics on it, it is necessary to find a way to use the keywords bitmaps. Since each bitmap correspond to one query wrote by one student in a specific goal of a topic, some consideration can be made, assuming that there is a specific function that sums the different bitmaps:

- by fixing the macro topic, the sum of the bitmaps produces a new bitmap with the keyword information of all the queries written by all the students who have done a workflow on a topic of that macro topic
- by fixing the topic, the sum of the bitmaps produces a new bitmap with the keyword information of all the queries written by all the students who have done a workflow on a specific topic
- by fixing the student, the sum of the bitmaps produces a new bitmap with the keyword information of all the queries written by that student in all his workflows, regardless the topic
- by fixing the topic and the student, the sum of the bitmaps produces a new bitmap with the keyword information of all the queries written by that student in the workflow corresponding to the topic
- by fixing the topic and the goal of the topic, the sum of the bitmaps produces a new bitmap with the keyword information of all the queries of that specific goal written by all the students who have done a workflow on a specific topic
- by fixing the topic, the student and the goal, the sum of the bitmaps produces a new bitmap with the keyword information of all the queries of that specific goal written by that student in the workflow corresponding to the topic

Since the solid is a NumPy array, there are some utility functions provided by NumPy to sum arrays, and I wrote two functions, reported in the code snippet 5.2, which helped me.

```
1 #function to sum the bitmaps in the provided solid
2 def sum_bitmaps(matrix):
3     while matrix.ndim > 1:
4         matrix = np.sum(matrix,axis = 0)
5     return matrix
6
```

```
#return the number of queries done in the solid provided in input
def sum_query(sub):
    if sub.ndim > 2:
        return sum(sum_query(sub[i]) for i in range(sub.shape[0]))
    else:
        return sum([1 if (np.sum(sub[x],axis = 0))>0 else 0 for x in
   range(sub.shape[0])])
```

Code 5.2: Sum Bitmaps functions

The first function takes as input an n-dimensional NumPy matrix and, if the matrix has more than one dimension, using NumPy's *sum* function the procedure compresses the matrix by one dimension. Eventually, the function returns a one-dimensional matrix that is the result of summing all the bitmaps of the n-dimensional matrix passed as a parameter to the function.

Algorithm 3 Sum Query Algorithm

It returns the number of queries done in the provided solid
Require: sub the solid pass as parameter
if $sub.ndim > 2$ **then**
  $counter \leftarrow 0$
  for i in $range(sub.shape[0])$ **do**
    $counter \leftarrow counter + sumQuery(sub[i])$
  end for
  return $counter$
else
  $counter \leftarrow 0$
  for x in $range(sub.shape[0])$ **do**
    sub[x] at this point is a bitmap
    if $sum(sub[x]) > 0$ **then**
      $counter \leftarrow counter + 1$
    end if
  end for
  return $counter$
end if

The second function, like the first, also needs the n-dimensional matrix as a parameter. It is more sophisticated because it has to manage empty bitmaps and it is recursive. Let explain in details how it works:

- **base case**: if the dimension of the solid pass as parameter is less or equal than 2 then the function consider the size of the first dimension and iterates over it. With this dimension the solid is basically a list of bitmaps, so the function iterates over them and for each bitmap it checks if it contains at

least one non-zero value. Since a SPARQL query must contains at least one of the keywords described in Subsection 5.2.4 thus a valid bitmap for a query must contains at least one non-zero value. Hence, at each iteration it increases a counter only if the bitmap has a non-zero value, and in the end return this counter that corresponds to the number of queries.

- **recursive case**: if the dimension of the solid pass as parameter is more than 2 the recursion is needed. In this case the the function breaks down the solid calling the *sum_query* function for each sub matrix of $(n-1)$-dimension and in the end it returns the sum of all the values returned by the recursive calls

Given that python combined with NumPy allows to do complex operations in a very compact way, I report in more detail how the *sum_query* function works in Algorithm 3.

6

Ground Truth

The second part of the thesis is devoted to the ground truth's creation. In our domain of interest, we have 24 different topics each with specific tasks that together form a workflow. Hence the goal was to create a ground truth for each workflow. Since this process requires a lot of effort, for the purpose of the thesis I narrowed the domain of interest only to the Movies macro-topic. To recapitulate, the Movies macro-topic has four different topics, that are:

- **Directors** exploratory search, six students worked on it
- **The Batman movies** exploratory search, four students worked on it
- **Horror Franchises** exploratory search, five students worked on it
- **Tv Series** exploratory search, six students worked on it

6.1 Creation of Ground Truths

Recalling that also the ground truth must be a valid workflow, meaning that it complies with the basic rule that prohibits to use objects or predicates of Wikidata if they are not discover before through the exploration or if they are not provided by the authors of the workflow as a starting point for the exploration.

It is important to distinguish between three different types of tasks requirements in the workflows and I will use as example the requirements of the Directors exploratory search in Figure 6.1:

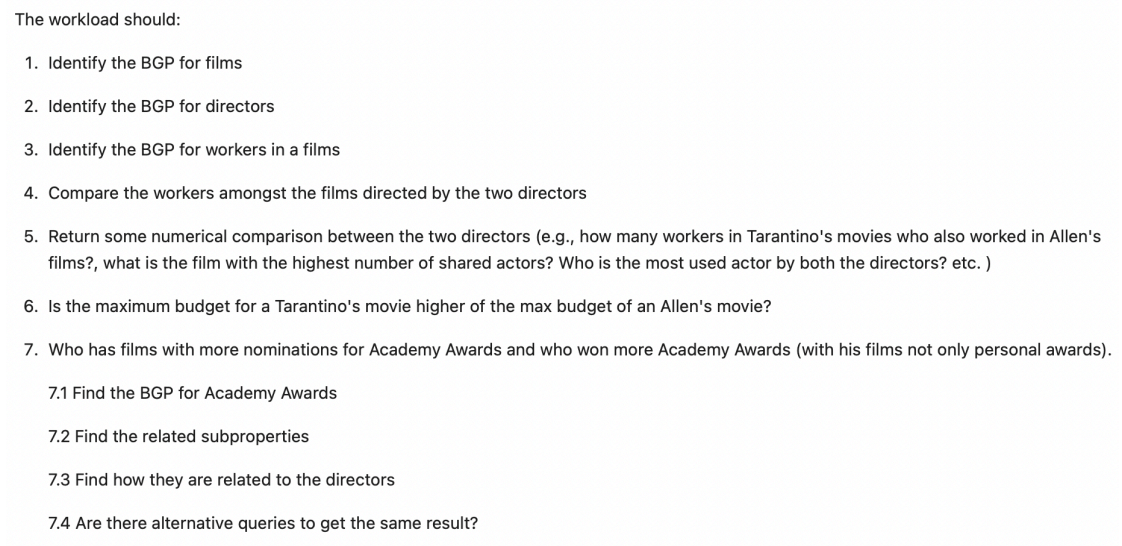


Figure 6.1: Example of the requirements of the Directors workflow

- **BGP** task: this type of task is very deterministic, meaning that there is a well defined answer for it. An example is the first one in Figure 6.1 which requires to identify the BGP for films. To answer this task correctly it is sufficient and necessary to write a query which returns in the result set the URI which represents the node film on Wikidata
- **deterministic** task: this type of task is also well define in terms of values to return, but they cover different kinds of requirements, for example the task 6 in Figure 6.1 requires to find the budget of the movies directed by Tarantino and Allen separately, pick the maximum value for the two directors and compare the value, hence the requirement is clear and also the answer. Furthermore the task 7 requires to compare the two directors in terms of Academy Awards won: it is therefore possible to find for each director the number of Academy Awards he won and compare the numbers
- **generic/exploratory** task: this type of task is not well defined as the other two types. An example is the task 5 in the Figure 6.1 which requires to explore the two directors and return some numerical comparisons. Although there are examples of comparison that a student can use as inspiration there is no guarantee that everyone will perform this task in the same way, hence the results of these tasks can be very different among students

After this clarification let's explain the ground truth creation process. Firstly, not to waste too much time creating my own workflow for each topic, I started reading and analyzing the notebooks of the students who worked on the topic. In this first phase it is possible to understand the domain of the topic, which are the main entities involved in the exploration and how they are related. Then

I proceeded writing the ground truth task by task. The goal of this job was to produce a well done workflow with possibly the shortest way to provide the true answer to the tasks. BGPs task are the easiest because almost all the students found them, although in very different way. The job here was to found the exploration which provided the smallest result set, thus the precision in the answer is bigger. Then, the queries which did not lead to new useful discovery of entities or predicates were removed from the notebook. Also for deterministic tasks the procedure was the same: starting from the work of the other students I tried to understand which were the ones who provided the most complete result set. In order to verify that, I benefited from the use of the Wikidata website and also web searching to cross checking what I found in our Wikidata snapshot. This led me to realise that our Wikidata snapshot lacked a lot of data that would provide an immediate response to the activities. Some students have put a lot of effort and they tried different ways to provide a good answer anyway. This often paid off big time because the missing data were some predicates so it was possible to getting around the problem and follow another route while on other occasions there were missing entities and in this situation nothing could be solved. The generic tasks were the most difficult because there was no exact solution. As I stated before, students faced this type of tasks in various ways providing very different results. For these type of tasks the only thing I could do was to analyze everything and report in the ground truth the two or three more interesting queries. Also in this cases I used the online Wikidata and the web searching.

The essential thing to keep in mind is that Wikidata is constantly being updated, so it is possible that data may not have been captured in our original snapshot or may have been added or updated after the time of the snapshot.

6.2 Storage Model of Ground Truths

Once the ground truths were created the results needed to be stored, in order to compare them with the students' workflows. This feature is very important and we will discuss about it later in Chapter 7.

In order to find the best structure to store the results the following considerations must be made:

- the results must be accessible and easy to read by another program

- the results must be separated somehow by their specific task, in order to associate a result to a specific task
- there must be a distinction between the type of the tasks in order to accurately assess student performances

After analysing the different possibilities, the choice was to use a JSON file. The JSON file associated to a ground truth has a description object, which is a string, that contains the title (topic) of the ground truth. Then, there is a complex object called *results* which contains all the information about the ground truth's results task by task. This object is a dictionary and each descendant object is identified with a key, that is the number of the task. Each task object contains three mandatory information which are:

- **type**: this element is related to the type of the task and it can be *single*, *referred* or *set*
- **check**: this element provide the type of the check that it must be done at the evaluation stage. I will go deeper afterwards
- **values**: this is the element that contains the results of the ground truth for the current task and it is always an array of other elements

The values array can contains several type of objects which can be Wikidata objects or predicates or simple literals like dates and numbers. Whenever I refer to a Wikidata object or predicate I am always referring to its URI because it is the only way to identify it, nevertheless every Wikidata object or predicate reported in the JSON file as solution of its task is always composed by its URI and its label that I will call *name*. The label of the objects is surely useful to the human understanding, but it is also useful at the evaluation stage, given that some students did not always report the URI of the objects but only their label. The more frequent shapes of the elements in this array of values are:

- Wikidata objects: they are elements composed by two fields, the *URI* and the related *name* (label)
- literal values: they are elements composed by the single field *value*
- literal values associated to Wikidata objects: they are elements composed by one literal stored in the field *value* which refers to a specific object. Hence these elements will also contain a field *refers_to* that usually is an URI which represents a Wikidata object and the field *refers_to_name* that is the Wikidata label associated to the URI in the *refers_to* field

- Wikidata objects associated to other Wikidata objects: they are elements composed by four fields: the *URI* and the related *name* (label) which refers to a specific object stored in the field *refers_to*. It also contains the *refers_to_name* for the textual representation of the referring object

The check field contains the information about the name of the field to check in each element of the values array. I have already presented the main types of the objects that the values array may contains and basically there are Wikidata elements or literals, so the check is usually performed against the *URI* when we are facing Wikidata objects or the *value* fields when they are literals. However, there are situation when the check field may contains more than one value hence the check field is an array. This situation often occurs when there are literals or Wikidata objects associated to others Wikidata objects because sometimes students did not used to return both the URI and the label in the result set but they only returned the labels without their URIs.

Lastly the type field provide information about the type of the task but to be more precise it contains useful information on the shape of the objects in the values array. I already stated that it can assume three different values and for a better understanding I will explain in detail the different situations.

6.2.1 Single Types

This value is usually used when the tasks require to find a BGP. In the Section 6.1 when I presented the BGP tasks I stated that a correct answer must contain the URI which represents the required element. Hence, the type of elements in the values array of these tasks are always Wikidata objects and the pivotal part is their URIs but for the sake of clarity I had always added also the associated label. In fact the check field in the BGP tasks will always contain only the value *uri* meaning that the valuable information for this task is the URI of the objects contained in the values array. There may occur situation when there are more than one valid URI that correctly answer the BGP tasks. To face these rare situations I added an optional field in the task JSON object called *any_all* which can assume only two values, *any* or *all*. The role of this variable is to define what a result set should contain in order to be correct. If the value of this variable is *all* it means that the task required to find all the elements contained in the values array thus to be correct a result set must provide all those elements. On the other hand, if the value of this variable is *any* it is sufficient that the result set contains only one of the elements listed in the values array.

(a) Single type object with "all" constraint

(b) Single type object with "any" constraint

Figure 6.2: Examples of single type objects in the JSON ground truth's file

Figure 6.2 shows two examples of these single type objects stored in the JSON ground truth's file. On the left, in the Figure 6.2a, there is the example when the *any_all* field is set to *all*. For this task the requirement was to find the BGP for *film*: the values array in fact contains the element with both the URI and label which represent a film in Wikidata. Since this type of task required a BGP the elements field to check is the *uri* that correspond to the value of the check's field. As mentioned before the JSON file contains also a description which reports the topic's name of this ground truth.

On the right, in Figure 6.2b, there is the opposite situation when the *any_all* field is set to *any* and the requirement was to find the BGP for directors. Since this workflow investigated movies there are two classes of Wikidata that represents directors. The more general is the *director* while the more precise in the movies context is the *film director*. Since both entities are correct for the meaning of this workflow, an answer will be considered valid if the result set provides at least one of these two classes. In fact the values array contains two elements that represent exactly the *film director* and the *director*.

6.2.2 Referred Types

These type of tasks are very frequent for those which required to find something associated to other values and the elements inside the values array are usually literals or Wikidata objects associated to another Wikidata object. The value of the *type* field is *referred*.

Figure 6.3 shows an example of this element's structure in the ground truth's file. In this workflow the exploratory scenario was to investigate horror movies and franchises and in particular they wanted to compare the Halloween franchise

```
▾ root:
    description: "Workflow's name: Movie Workflow Series ("Horror Franchises explorative search")"
  ▾ results:
    ▾ 2:
        type: "referred"
      ▾ check: [] 2 items
          0: "uri"
          1: "name"
        elements_per_tuple: 1
      ▾ values: [] 2 items
        ▾ 0:
            uri: "http://www.wikidata.org/entity/Q314914"
            name: "Donald Pleasence"
          ▾ check: [] 2 items
              0: "uri"
              1: "name"
            refers_to: "http://www.wikidata.org/entity/Q221103"
            refers_to_name: "Halloween"
        ▾ 1:
            uri: "http://www.wikidata.org/entity/Q2865227"
            name: "Betsy Palmer"
          ▸ check: [] 2 items
            refers_to: "http://www.wikidata.org/entity/Q1243029"
            refers_to_name: "Friday the 13th"
    ▸ 3:
    ▸ 4:
    ▸ 5:
    ▸ 7:
```


Figure 6.3: Example of referred type object in the JSON ground truth's file

with the Friday the 13th. There were given the URIs of two movies, one for each franchises, as a starting point for the exploration. This task required to find who was the most famous actress (or actor) on Halloween and Friday the 13th (the movies initially given) at the time of the release. Hence, it was expected that students provided a result set consisting of two rows with two objects each: the movie and the most famous actor who acted on that movie at the time of the release. Indeed, in the values array we can see two objects: the first one contains the URI and the label of Donald Pleasence who refers to the Halloween movie, while the second one contains the URI and the label of Betsy Palmer who refers to the Friday the 13th movie.

Since this type of task requires two checks to state the correctness for each element of the answer, there is another check field for each element in the values array. This was added to allow the containment of different types of values associated to Wikidata objects and the Figure 6.4 clearly shows this situation. We are talking about the same workflow on the Horror Franchises exploratory search and the mentioned task required firstly to find the entire list of the movies

of both the Halloween and Friday the 13th franchises, and eventually to return for each of those film the director and the year of publication. Thus the expected format of the result set was a list of elements which were supposed to contain the movie, the director and the year of publication. This type of requirement is slightly more sophisticated to handle because there are several elements that can be literals (the year of publication) or Wikidata objects (the director) associated to the main Wikidata object. To solve this specific problem, a special object could have been created that contained these two pieces of information, but it would have been very difficult to handle at the evaluation stage. There was therefore a need to organise everything as general as possible.

Figure 6.4: Example of different type of values in a referred object's task of the JSON ground truth's file

To achieve this, when there is a referred type task object, each element inside the values array must contain:

- **refers_to** and **refers_to_name** fields which represent the referring Wikidata object

- **check** field which basically specifies the type of the reference object as this field can usually be an array whose elements are *uri* and *name* for the Wikidata objects, or the string *value* for the literals
- the couple **uri** and **name** fields if there is a Wikidata object, or the **value** field if there is a literal

Furthermore, there is another essential parameter: the **elements_per _tuple**. Given the need to unbundle complex result sets in single pieces, in this example we had separated the year from the director, one must somehow know the number of associated objects to the referred Wikidata object. Hence, I added this field to keep this information and use it at the evaluation stage.

6.2.3 Set Types

Finally there are the tasks where the result set is not well defined because the requirement is more general and the results can differs one from another or the result set comprises many elements thus it is not possible to use the *single* type described in the Subsection 6.2.1.


```
▾ root:
    description: "Workflow's name: Movie Workflow Series ("Tv series explorative search")"
  ▾ results:
    ▸ 1:
    ▾ 2:
        type: "set"
      ▾ check: [] 2 items
          0: "uri"
          1: "name"
      ▾ values: [] 6 items
        ▾ 0:
            uri: "http://www.wikidata.org/entity/Q200566"
            name: "Cobie Smulders"
            value: "145"
        ▸ 1:
        ▸ 2:
        ▸ 3:
        ▾ 4:
            uri: "http://www.wikidata.org/entity/Q199927"
            name: "Alyson Hannigan"
            value: "143"
        ▸ 5:
    ▸ 4:
    ▸ 5:
```


Figure 6.5: Example of set type object in the JSON ground truth's file

The example shown in Figure 6.5 refers to the workflow about television series. The scenario was to explore the television series "How I met your mother",

investigate the main aspects and compare it with "The Office". In particular the task highlighted in the figure required to find the most present actors, thus the result set must be a list of actors who acted on "How I met your mother". The result here reports the six most famous actors given that the request was vague on the size of the result set. The meaning of this type of object is to provide the largest and most accurate set of result that possibly should be a subset of a result set provided by the students doing this workflow. As for the other types of object task there is the *check* field that specifies the field to check in order to the determine the correctness of the answer.

6.3 Storing Process of Ground Truths

After the detailed illustration of the JSON model containing all ground truth information of a given topic let's explain how the storing process works. First of all this process is not automatic, so there is not a script which read the python notebook of the ground truth and automatically produce the ground truth JSON file. This is due to the fact that there are tasks which are not reported in the JSON file because of their complexity of the result set and their very ambiguous requirement. For example, a task in the Horror Franchises exploratory search required to investigate the workers and check any commonalities between the two movies Halloween and Friday the 13th: the requirement is not precise and the answers to this task can be very different. Someone could compare the size of the cast members, someone can check if there are common workers, others can compare the directors, or the composers, so there is no proper answer to this task.

The workflow of each ground truth, before the exploration that will provide the solution to the requirements, begins with some utility functions and some variable definitions that will help during the storage process. They are:

- find the indexes which identify the current topic. Recalling the index maps mentioned in Section 5.3, one was devoted to store the index pairs that identify a topic. The first function reads that table and retrieves the correct index pair for the current ground truth workflow. These indexes are needed because the ground truth JSON files are stored in a folder called **result** and each ground truth's file is named *workflow_i1_i2* where *i1* is the macro topic index and *i2* is the topic index
- define some global constant string variables that are the main keywords to use in the JSON file

- define the global variable which represents the JSON object
- define the function to add a new task object in the result dictionary

At this point whenever a task needs to be inserted in the JSON ground truth I manually created that object and I called the function to add the result of the current task to the global object that will also store it on the disk. Depending on the type of task, there are different functionalities, but in general, the following operations are required to add a new task to the ground truth:

- define the **type** of the task amongst *single*, *referred* and *set*
- create the array of objects to put in the **values** array
- define the check field according to the type of the task

The function will subsequently create a dictionary object with the input parameters just described above and put this object in the global JSON under the key of the current task, also passed as parameter.

7 Evaluation and Analysis

Once the ground truths are created and the results are stored, it is possible to evaluate the student workflows in order to check their results. Evaluation is only possible for movie-related workflows given that the ground truths were created only for those topics. The evaluation and keyword analysis are performed in parallel and produce a single file containing the entire notebook analysis.

7.1 Workflows Evaluation

When a ground truth file exists for a topic, the script evaluates the notebooks of students who have worked on that topic.

The evaluation, shown in Algorithm 4, requires some functions and objects:

- notebook's file path
- ground truth's file path
- a function which, given the student's results and the ground truth limited to a specific objective, compares them and verifies whether the student answered that task correctly
- the functions already described in Chapter 5 which find the goals of workflow, collect the queries and run them

First of all the function retrieves the goals of the topic and subsequently collects the queries of the notebook. Finally it loads the ground truth from the disk. In order to compare the result of the ground truth with the once provided by the student, it is necessary to run the student queries. Thus, for each goal

Algorithm 4 Evaluation Algorithm

Require: nb a notebook file path
Require: $gtPath$ a ground truth json file path
Require: $verifyResult$ a function that given the ground truth and a result set checks if the result set contains elements that correspond to the ground truth

```
goals ← findWorkflowGoals(nb)
queries ← collectQueries(nb, goals)
gt ← open(gtPath)
eval ← {}
for g in goals do
    solutions ← []
    actualGt ← gt["results"][g]
    index ← 0
    for q in queries[g] do
        index ← index + 1
        x ← runQuery(q)
        check ← verifyResult(actualGt, x)
        if check > 0 then
            precision ← computePrecision()
            recall ← computeRecall()
            solutions.append({index, precision, recall})
        end if
    end for
    if solutions is not empty then
        eval[g] ← findBest(solutions)
    else
        eval[g] ← {0, 0, 0}
    end if
end for
return eval
```

of the topic, the function initially retrieves the result from the ground truth for that task and then it starts to execute the student queries. The output of a query is a list, and each element of the list is another list of n-tuples depending on the number of variables included in the selection. An example of a query with its output is shown in Figure 5.2: the query's select indicates that there are two different variables for each result row *p* and *pName*. In fact, in the output cell there are several lists with two tuples each. Then, each tuples represents the name and the value of a variable. In this example the query returns the URI and the label of some predicates of the television series "How I met your mother". Proceeding with the algorithm, the result set of the query is stored in a variable

which is used in the *verifyResult* function. This function is devoted to compare a generic result set with a ground truth result set and it always returns:

- **0** if the result set provided has no correspondence with the ground truth
- **positive integer** which depends on the number of correspondences between the result set provided and the ground truth

To count the number of correspondences there are three different algorithm, one for each type of the task described in Section 6.2. They are slightly different given the different shape of the objects in the values array which depends from the task type. Algorithm 5 shows the function to return the number of correspondences for the referred type tasks. The function starts declaring the variable matches that will count the number of correspondences and eventually be the returned value of the function. It also defines an empty list called *foundEl* which represents the elements already found during the procedure: this list helps to avoid counting the same element twice. Each element of the result set must be compared with the ground truth elements. Before comparing it, since the initial row element of the result set is a list of tuples, where each tuple contains both the name and the value of the variable, a first cleaning operation is performed in order to create another list (called *elements*) which contains only the value of the variables, given that the name is not essential. Then, for each element in the ground truth:

- store the value of the fields *refers_to* and *refers_to_name* in two variables
- verify the presence of the referred object in the *elements* list. Since the shape of the elements in this type of task are Wikidata objects or literal which refers to another Wikidata object, to provide a valid answer the referred object must be reported in the result set
- verify the presence of the other object. If this is a Wikidata object, then the *check* field in the ground truth is a list which contains the two elements *uri* and *name*. This means that the *uri* of a Wikidata object or its label must be in the *elements* list. Otherwise if it is a literal, the function checks its presence in the *elements* list.
- when a new element is found, meaning that both the object and its reference are in the *elements* list, the *matches* counter and the list of elements found are updated. A tuple with all the information about the object and its reference is created and it is appended to the list of elements found. During the subsequent iterations before counting a new match this list must be checked

Algorithm 5 Referred Type Check Algorithm

Require: $trueResult$ a json object represent the task's ground truth
Require: $resultSet$ a query result set
  $matches \leftarrow 0$
  $foundEl \leftarrow []$
  $multRes \leftarrow trueResult["values"]$
  for res in $resultSet$ **do**
    $elements \leftarrow [t[1]$ for t in $res]$
    for val in $multRes$ **do**
      $ref\_uri \leftarrow val["refers\_to"]$
      $ref\_name \leftarrow val["refers\_to\_name"]$
      if not (ref_uri in $elements$ or ref_name in $elements$) **then**
        continue
      end if
      if type($val["check"]$) == list **then**
        $t\_u \leftarrow [val[j]$ for j in $val["check"]]$
        $tuples \leftarrow [(x, y)$ for x in t_u for y in $elements]$
        for t in $tuples$ **do**
          $tmp \leftarrow (ref\_uri, ref\_name, t[0], t[1])$
          if tmp **not in** $foundEl$ **and** $t[0] == t[1]$ **then**
            $matches \leftarrow matches + 1$
            $foundEl.append(tmp)$
          end if
        end for
      else
        $v \leftarrow val[val["check"]]$
        $tmp \leftarrow (ref\_uri, ref\_name, v)$
        if tmp **not in** $foundEl$ **and** v in $elements$ **then**
          $matches \leftarrow matches + 1$
          $foundEl.append(tmp)$
        end if
      end if
    end for
  end for
  return $matches$

The basic concept is the same for the other two functions, but this is the most complete because it contains the reference object which is not present in the others.

After verifying the result of the current query with the ground truth, if the function returns 0 it means that the query did not contain any useful element. On the other hand if the function returns a positive integer it must be computed the precision and the recall of the result, in order to update the evaluation of the notebook. Since the recall is obtained by dividing the number of relevant items retrieved over the total number of relevant items, these two values are respectively the number of matches (returned by the *verifyResult* function) and the size of the values array of the current task in the ground truth. There is a special case when the type of the task is **single**, because the value of the *any_all* field plays a crucial role: if the value is *any* then the recall will be always 1 if the *matches* value returned by the *verifyResult* function is positive.

The precision is defined as the number of relevant retrieved items, that in our context is the *matches* value returned by the *verifyResult* function divided by the size of the query result set. Also for the precision there is a special case that occurs when the type of the task is **referred** because the value of the *elements_per_tuple* field plays a crucial role: since this field defines the number of elements that are associated to the referred Wikidata object, one assumes that for each row element in the result set there is the referred object with all its associated objects. Hence, the standard recall value must be divided by the value of the *elements_per_tuple* field, otherwise there will be unfeasible precision values greater than 1.

Each result set with valuable information, meaning that elements in the result set are also in the ground truth thus the precision and the recall for this result set are greater than 0, is considered. In fact in the Algorithm 4 for each goal of the workflow there is a solutions array that is populated when a query produces a good result set. The elements in this array are dictionaries which contain the information about the recall, the precision and the index of the query, useful to understand after how many queries the student found the result. When all the queries of the current goal are executed the evaluation object must be updated and two situations can occur:

- the solutions array is empty, meaning that for the current goal the student did not provide any query which produced a valuable result set. Hence, in the evaluation object the precision and the recall for the current task are both 0

- the solutions array contains at least one element, meaning that the student provide some good result set for the current goal. If there are several elements, the best is chosen according to the largest sum of precision and recall. If there are solutions with the same sum of precision and recall, it is chosen the one with the highest recall

7.2 Workflows Analysis

In parallel with the evaluation the script also performs the keyword analysis. Algorithm 6 shows the procedure that creates an object which represents the usage of the SPARQL keywords in a notebook. In order to execute this analysis, some functions and parameters are required:

- notebook's file path
- workflow 's topic
- a function (called *doStatistics*) that accept four optional parameters: the macro topic, the topic, the student and the goal. This multi-purpose function allows to cut different portion of the solid that collects the keyword bitmaps of all the notebooks. If no parameters are used, it sums the bitmaps of the entire solid. Whenever a parameter is set, the function delimits the corresponding part of the solid and only sums the bitmaps in this portion. As a result this function always returns a bitmaps sum on a specific portion of the solid depending on the parameters. In this particular stage, we are interested in the usage of the keywords in each goal of the workflows student, thus the parameters used in the function are the topic, the student and the goal. The macro topic is automatically obtained by the topic. The analysis also provide an overall usage of the keywords and to obtain it, it is sufficient to use the *doStatistics* function providing only the topic and student parameters
- a function (called *sumQuery*) which, given a specific portion of the solid returns the number of queries. I have already discussed this function in the Algorithm 3
- the function which finds the workflow goals given a notebook described in Subsection 5.2.2
- the function devoted to queries collection described in Subsection 5.2.3
- all the utility functions described in Section 5.3 that given the student number, the topic, the goals returns their indexes in the solid

Algorithm 6 Keyword Analysis Algorithm

Require: nb a notebook file path
Require: $topic$ of the workflows
Require: $keywords$ array that contains all the SPARQL keywords
Require: $findWorkflowGoals$ function
Require: $collectQueries$ function
Require: $doStatistics$ a function that given the topic, the student and the goal, sums the keyword bitmaps
Require: $sumQuery$ a function that returns the number of queries in a specific portion of the solid
Require: all the utilities to get the indexes

```
goals ← findWorkflowGoals(nb)
queries ← collectQueries(nb, goals)
statistics ← {}
iMacro ← getIndexByMacroTopic(getMacroByTopic(topic))
iTopic ← getIndexByTopic(topic)
stud ← getStudentId(nb)
iStud ← getIndexByStudent(stud)
goals[""] ← "Empty"
for g in goals do
    iGoal ← getIndexByGoal(goals, g)
    stat ← doStatistics(topic, stud, g)
    totQ ← sumQuery(solid[iMacro, iTopic, iStud, iGoal])
    goalStat ← {"number" : g, "description" : goals[g], "queries" : totQ}
    goalStat["keywords"] ← {}
    for i in range(len(keywords)) do
        goalStat["keywords"][keywords[index]] ← stat[index]
    end for
    if g!= "" then
        statistics[g] ← goalStat
    else
        statistics["0"] ← goalStat
    end if
end for
stat ← doStatistics(topic, stud)
totQ ← sumQuery(solid[iMacro, iTopic, iStud])
goalStat ← {"description" : "overall", "queries" : totQ, "keywords" : {}}
for i in range(len(keywords)) do
    goalStat["keywords"][keywords[index]] ← stat[index]
end for
statistics["overall"] ← goalStat
goals.pop("")
return statistics
```

The procedure starts, finding the goals of the workflow, collecting all the queries and converting the macro-topic, the topic and the student to the related indexes. It also declares an empty dictionary called *statistics* that at the end of the algorithm will be returned. Then the function analyses each goals of the workflow as follows:

- it converts the goal to the index
- it calls the *doStatistics* function to get the keywords usage for the current goal
- it calls the *sumQuery* function to get the number of queries of the current goal
- it creates a dictionary object called *goalStat* which contains information about the number of the goal and its textual requirement, the number of the queries wrote by the student for the goal and a keywords usage dictionary that is populated using the bitmaps sum returned by the *doStatistics* function
- it puts the *goalStat* dictionary in the statistic dictionary using the number of the goal as key

After having analysed all the goals the function repeats the same routine for an overall analysis of the notebooks. As before, it calls the *doStatistics* function without specifying a goal, so the function will provide the sum of the bitmaps of the entire student's notebook. It calls the *sumQuery* function and it populates a *goalStat* dictionary containing the number of the queries, the keywords usage and as description the string *overall*.

7.3 Evaluation and Analysis Merge

The parallel processing of the evaluation and the keywords analysis of the notebooks allows to merge them and store everything in a single file. This final procedure groups all the functionalities shown, to produce for every student's notebook, a file which contains information about the keywords usage and the evaluation. The result of the Algorithm 7 execution will be a folder containing as many folders as there are students who have worked on this project. Hence, the name of each sub-folder corresponds to the student's identification number and the student's folder will contain a json file for each workflow he or she has worked on. The requirements of the Algorithm 7 are:

- *workflows* dictionary created by Algorithm 1, which contains the list of the notebooks file path grouped by their topic and macro topic
- *evaluateNotebook* function shown in Algorithm 4, which given the ground truth, the goals of the topic and a notebook, creates an evaluation for each goal of that notebook
- *computeNotebookStats* function shown in Algorithm 6, which given the topic, the goals of the topic and a notebook returns the keywords usage in the different goals
- all the utility functions described in Section 5.3
- *evalDir* the path where the produced files will be stored
- *resultDir* the path where there are stored the ground truth json files

First, the algorithm checks whether the directory provided in the *evalDir* variable exists, and if not, creates it. Then it starts iterating topic by topic. As usually it converts the topic and the macro topic in the correspondent indexes to use later, and it collects the workflow's goals. If in the result directory, the one dedicated to store the ground truth files, exists a file for the current topic, it loads the file and it stores in a variable the results field. Recalling the model of the ground truth's file described in Section 6.2 there are two main fields: *description*, that reports the topic of the ground truth, and *results* which contains all the true result set goal by goal. At this point for each notebook of the current topic the procedure is:

- get the student id number
- create the student folder under the evaluation folder it it not exists
- create a dictionary object (called *obj*) which in the end represents the json object to store for the current notebook with the main information about the macro topic, the topic, the student and the notebook file path
- if the ground truth for this topic exists, run the evaluation on the notebook and store the result in a variable at the moment
- compute the statistics on the keywords usage in the notebook and store the result in a field called *goals* of the *obj* dictionary
- merge the information about the evaluation adding for each goal a field *evaluation*
- store the *obj* dictionary in a json file in the student folder, following the standard *workflow{iMacro}_{iTopic}.json* where *iMacro* is the index of the macro topic and *iTopic* is the index of the topic in the solid

Algorithm 7 Complete Analysis and Evaluation Algorithm

Require: $workflows$ the dictionary which contains all the workflows file path
Require: $evaluateNotebook$ the function shows in the Algorithm 4
Require: $computeNotebookStat$ the function shows in the Algorithm 6
Require: all the utilities to get the indexes
Require: $evalDir$ a path to the directory to store all the results in json files
Require: $resultDir$ a path to the directory where are stored the ground truth json files

```
if evalDir not exists then
    createDir(evalDir)
end if
for macro in workflows.keys() do
    for topic in workflows[macro].keys() do
        iMacro ← getIndexByMacroTopic(macro)
        iTopic ← getIndexBySubTopic(topic)
        goals ← findWorkflowGoals(workflows[macro][topic][0])
        trueResult ← {}
        workPath ← "workflow" + iMacro + "_" + iTopic + ".json"
        path ← resultDir + workPath
        if path exists then
            trueResult ← json.load(open(path))["result"]
        end if
        for nb in workflows[macro][topic] do
            stud ← getStudentId(nb)
            evalStud ← evalDir + stud
            if evalStud not exists then
                createDir(evalStud)
            end if
            obj ← {"macro" : macro, "topic" : topic, "student" : stud}
            if trueResult not empty then
                evaluation ← evaluateNotebook(nb, goals, trueResult)
            end if
            obj["goals"] ← computeNotebookStats(nb, goals, topic)
            obj["filepath"] ← nb
            for key in evaluation do
                if key not in obj["goals"] then
                    obj["goals"][key] ← {}
                end if
                obj["goals"][key]["evaluation"] ← evaluation[key]
            end for
            evalStudPath ← evalStud + workPath
            store(obj,evalStudPath)
        end for
    end for
end for
```

An example of file produced by this algorithm is shown in Figure 7.1. As stated before there are the macro topic, the topic, the student and the filepath fields. The goals field is another json object which contains all the goals identified by their number plus the overall object which is the sum of the keywords statistics of all the goals. Each goal's object contains the number, the textual requirement and the number of queries wrote to solve the current goal. The keywords object contains the information of the appearances of the keywords in the queries of the current task. Eventually, the evaluation object shows the student's best result for the current goal, compared with the ground truth.

```
▼ root:
    macro_topic: "Movie"
    topic: "Movie Workflow Series ("Directors explorative search")"
    student: "12345"
  ▼ goals:
    ▶ 0:
    ▶ 1:
    ▶ 2:
    ▶ 3:
    ▶ 4:
    ▶ 5:
    ▶ 6:
    ▶ 7:
    ▼ 7.1:
        number: "7.1"
        description: "Find the BGP for Academy Awards"
        queries: 9
      ▼ keywords:
          SELECT: 9
          ASK: 0
          DESCRIBE: 0
          CONSTRUCT: 0
          NESTED QUERY: 0
          DISTINCT: 1
          LIMIT: 0
          OFFSET: 0
          ORDER BY: 3
          FILTER: 1
          REGEX: 1
          AND: 0
          UNION: 0
          OPT: 0
          GRAPH: 0
          EXISTS: 0
          NOT EXISTS: 0
          MINUS: 0
          COUNT: 2
          MAX: 0
          MIN: 2
          AVG: 0
          SUM: 0
          GROUP BY: 0
          HAVING: 0
          GROUP_CONCAT: 0
      ▼ evaluation:
          recall: 1
          precision: 0.5
          query_num: 9
    ▶ 7.2:
    ▶ 7.3:
    ▶ 7.4:
    ▶ overall:
    filepath: "/home/ubuntu/AnalyticalWorkload/notebook/sparqlthesis/notebook/M12345/7fb80672bf.ipynb"
```


Figure 7.1: Example of a json file stored

8

Statistics

After collecting all the necessary data, organising them in order to group them, when necessary, by macro-topic, topic or student and defining the functions to obtain the usage of the keywords in the queries and the two functions to sum bitmaps and to get the total number of queries given a portion of the solid, it is possible to do some statistics.

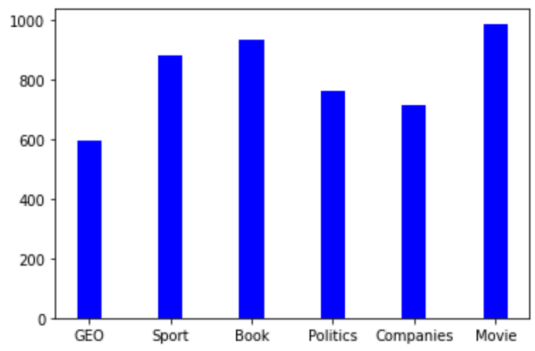
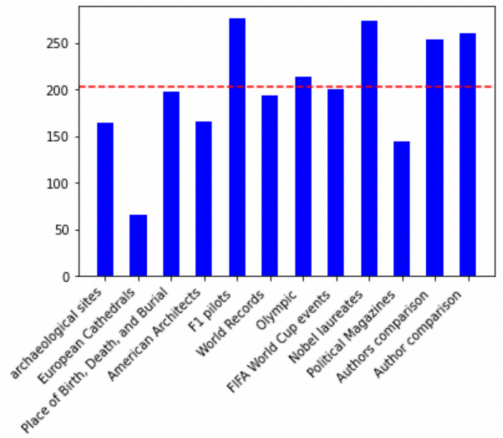


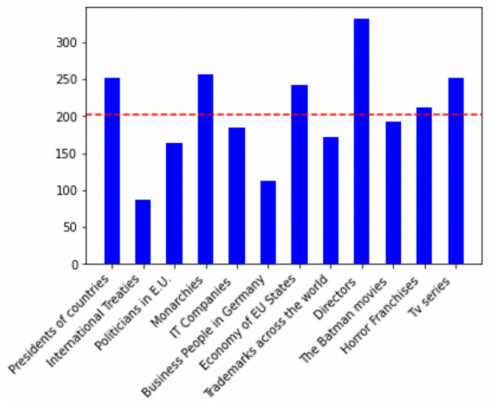
Figure 8.1: Numbers of queries wrote by students grouped by macro topic

Firstly we consider the statistics about the macro-topics (Figure 8.1). As reported in Table 4.1 there are 21 workflows for each macro-topic, hence they are perfectly balanced. It is clear that the Geography macro-topic is the one with the lowest number of total query. In contrast, movie-related topics were the ones with the highest number of queries. This can be the result of two factors: first the goals of the Geography-related topics were easier but since all the workflows were well-balanced (workload meaning) it is likely that Wikidata treats films in a more complex way.

In Figure 8.2 there are the total number of queries grouped by the different topics. On the left (Figure 8.2a) there are the topics about Geography, Sport and Book while on the right (Figure 8.2b) there are the topics about Politics, Companies and Movies. The red dashed horizontal line indicates the average



(a) Number of queries on Geography, Sport and Book

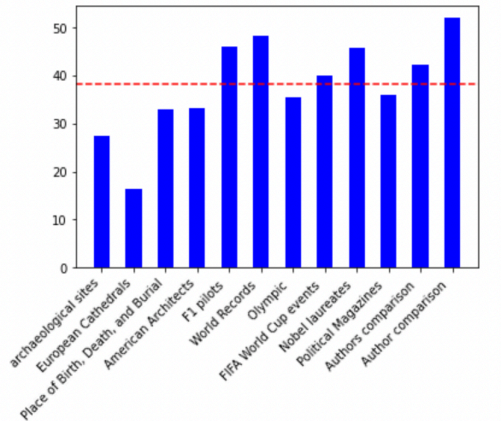


(b) Number of queries on Politics, Companies and Movies

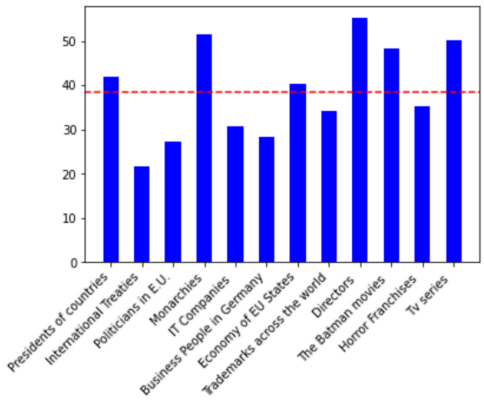
Figure 8.2: Comparison on the total number of queries by topic

number of queries for each topic, considering the total number of queries wrote by all the students in all the topics, which are **4861**, divided by the number of the topics, which are **24**. It can be clearly seen that all the Geography's topic (the four on the left in the Figure 8.2a) are below the average number of queries while the Movies topic (the four on the right of the Figure 8.2b) are all above the average except for one, although slightly, with a peak on the Directors topic. However, the values reported in these plots are the absolute values, considering the sum of all the queries did by all the students on a specific topic. Recalling that there are topics that fewer people have worked on, it would be better to consider a plot where the total number of queries is normalised by the number of people who worked on the specific topic. For instance, let consider the Movie macro-topic and look in more detail at the values. The Table 4.1 denotes that six students worked on Directors topic, four students worked on The Batman's topic, six students worked on Horror Franchises topic and five students worked on Tv Series topic. If we assume all the topics were well workload balanced, then it makes sense that the number of queries on Directors topic are about 50% more than the Batman movies topic given that there are two more people worked on it.

The two plots in Figure 8.3 represent the average number of queries wrote by topic. For example the total number of queries wrote on the Directors topic are 331 while the queries on The Batman movies topic are only 193. Given that six students worked on the Directors the value in the plot of Figure 8.3b is



(a) Normalized number of queries on Geography, Sport and Book



(b) Normalized number of queries on Politics, Companies and Movies

Figure 8.3: Comparison on the total number of queries by topic normalized on the number of students worked on the topic

almost 55 (331 divided by six) while the value of the Batman movies is almost 48 (193 divided by four), since only four people worked on it. This different point of view shows that there are topics that were apparently "bad", given the low number of total queries, that are actually better than many others. One example is The Batman movies against the Horror Franchises. In the plot of Figure 8.2b the Horror Franchises was above the average number of queries per topic while The Batman was below the average. The plot of Figure 8.3b reverses the situation. The same happens in the Sport macro-topic between the topics World Records and Olympics.

The other statistic is based on the use of SPARQL keywords, which depending on the type of topics could be very different. First of all I did a global analysis of the keywords over all topics and notebooks. The results in Table 8.1 shows that there is a predominance of the `Select` keyword with a utilisation of over 99%. Selection is basically always used because the topic requirements were always to provide answers involving the Wikidata objects provided as a starting point. In order to do this, students approach was to explore the Wikidata objects returning their properties which led to new discoveries. A small percentage of queries used an `Ask` form to verify if some patterns exists or not, because students usually did a select on that pattern and checked the results: if the result set was empty, then it would be the to run an ask query with a negative answer. But if the result set was not empty, then probably there were something interesting in that result, because they were doing exploratory search and almost always they

wanted to see the the concrete result and not a simple true or false. `Ask` queries were also used in some few tasks that required an answer of a comparison (e.g. Is the maximum budget for a Tarantino's movie higher of the max budget of an Allen's movie?).

Keyword	**Occurrences**	**Percentage**
Select	4841	99.58%
Ask	51	1.04%
Describe	1	0.02%
Construct	1	0.02%
Nested query	416	8.55%
Distinct	3412	70.19%
Limit	2349	48.32%
Offset	0	0.0%
Order by	1863	38.32%
Filter	1487	30.59%
Regex	485	9.97%
And	8	0.16%
Union	259	5.32%
Optional	343	7.05%
Graph	6	0.12%
Exists	171	3.51%
Not exists	137	2.81%
Minus	8	0.16%
Count	1293	26.59%
Max	114	2.34%
Min	138	2.83%
Avg	57	1.17%
Sum	29	0.59%
Group by	1050	21.6%
Having	56	1.15%
Group_concat	201	4.13%

Table 8.1: Keyword usage out of a total of 4186 queries

The second more frequent keyword is the `Distinct`. It is not uncommon that queries can produce a result set with some repetitions. Sometimes this is what we need, some others this is very annoying because it messes up the result. In these cases, the `Distinct` keyword is used in the `Select` to avoid repetition of rows in the result set. Let's do a practical example. Consider a film director and all the film he has directed, we want to find the properties of those films. First of all we had to bind the director with all his/her movies and then return all the

outgoing edges of those movies. Usually movies has some standard properties such as the instance of, the title, the genre and many others. If we are only interest in discovering the different properties of a generic film, returning all the properties of all the director's movies will produce a result set with a lot of duplicates. This can also happen when there are properties with several objects, for example a person can be educated at different Universities, thus the property *educated at* appears at least twice in the result set. Doing exploratory search a fair number of queries is devoted to understand and discover relations between objects, hence many times we just want to know if there is a relationship and how it is called, cardinality is not of interest.

In order of frequency we find the `Limit` keyword, used in more or less half of the queries. `Limit` allows to cut the result to a certain number of rows and it is very useful when we want to see some instances of a class just as example to understand which elements are in that class. Sometimes `Limit` is used when we know that a result set is not so small, and we hope to find what we are looking for in the first rows. In particular, `Limit` was crucial in our behaviour. Given that there were more students working on the same database, complex queries with big result sets slows down the performance, thus students were asked to use the `Limit` every time they were not sure about the size of the result set. Furthermore, it is very easy to write a wrong query that will retrieve millions of rows as result.

Many workflow tasks required aggregation. These tasks are the most interesting because they usually summarise previous findings. The `Group By` keyword in fact was used in more than 20% of the queries while the most used aggregation function is the `Count` that plays a role also in other situation outside the `Group By` operation. Also the `Order By` is very used, more than one query over three used it, and this keyword is usually used in the `Group By` queries, in order to rearrange the result set and then possibly cut it with a `Limit`. The ordering of the output was also used in the queries which retrieve the properties of the objects in order to search alphabetically.

`Filter` is the last very used keyword. Its use is necessary to remove data from the results. A very common example of its utilization regards dates, in particular when only certain time intervals need to be selected.

After this generic analysis of the whole dataset, it is interesting to look at each of the macro-topic individually. Firstly, to check the distribution of keywords and whether their use is balanced between the different macro-topics. Then,

to understand what the key differences were between the macro-topics, e.g. whether some required more aggregation than others. Table 8.2 shows the keywords usage for each macro-topics, but many underused keywords have been omitted.

In general, we can see that `Select` is used in more than 99% of the keywords for each macro-topic, as already indicated by the overall statistics in Table 8.1. The tables also show the use of `Ask`, the second most common SPARQL construct, which was used most often in the macro topics of Movies, Companies and Politics. This is probably due to the fact that there were some tasks in the workflows of these macro topics that required a true/false answer. One of the examples already used is a task in the Director's Workflow of the Movie Macro topic: "Is the maximum budget for a Tarantino's movie higher of the max budget of an Allen's movie?". Given that six students worked on this workflow, if we assume that everyone used an `Ask` query to answer this task, we have already reached half of the usage of this keyword. Other uses may be other tasks of the same type or student curiosity.

Another keyword very used and well balanced amongst macro-topics is `Distinct`. In each macro topic, the use of this keyword was around 70%, a reminder that exploratory research requires many queries to understand the properties of the entities involved and their relationships.

One interest statistics is the usage of `Limit`. There are macro-topics, such as Movie and Book, which use this keyword in 40% of their queries, while other macro-topics, such as Geography and Companies, use this keyword in 55-60% of their queries. A plausible motivation is that there is more data available for these macro-topics and therefore larger result sets. Another motivation is that there are more tasks on these macro-topics workflows which required a cut on the result set. For example, questions such as "Return the top-10 of..." certainly require the use of the `Limit`.

In terms of aggregations, which are certainly the most interesting queries to write and visualise their results, there is a lot of diversity in the data of the different macro-topics. The main keyword concerning aggregation is Group By. Book's macro-topic is the one with the lowest usage of this keyword, i.e. the one with the lowest aggregation. Hence, it is clear that for this macro-topic there were fewer tasks requiring aggregation or it is possible that the entities involved in this macro-topic are related in a simpler way. On the other hand, the macro topic Companies is the one with the most aggregations. More than

every third query uses the keyword `Group By`. This may be due to the complex relationships between the entities involved in this macro-topics or to the more complex requirements of the tasks.

`Count` is very used in the aggregation query as aggregation function. The other aggregation functions (`Max`,`Min`,`Sum`,`Average`) in fact are omitted in these Tables because their low usage is not comparable to `Count`. Actually, `Count` is at least as common as `Group By`. Although it is used as much as `Group By` in Companies, it is used much more in the other macro-topics. This difference may be due to the fact that in the other macro-topics there was a greater need to know the count of other things, e.g. the number of properties of certain entities. Or, in the Companies, the other aggregation functions were also used a lot.

About `Order By`, it seems that its utilization is proportional to `Group By`. In fact, many aggregation queries require sorting by the aggregation function to get the best results first and eventually truncate the result up to a certain size. Although this is one of the most common uses, sorting is useful when searching for the label of a specific property, as entities usually have hundreds of properties.

Another type of queries proportional to the use of `Group By` are the nested query. Even though this is not a SPARQL keyword, we have included it in the statistics because it is very important to understand when queries become complex. In fact, the aggregation queries, that are the most useful and interesting to perform, are also the most complicated, because they usually relate many entities from different domains. Very often, these queries require aggregation on a result set that cannot be obtained immediately, but a sub-query is required.

Geography Macro Topic		
Keyword	**Absolute**	**Relative**
Select	590	99.66%
Ask	3	0.5%
Nested	43	7.26%
Distinct	439	74.15%
Limit	353	59.62%
Order by	207	34.96%
Filter	138	23.31%
Count	195	32.93%
Group by	151	25.5%
TOTAL	**592**	**100%**

(a) Geography keywords usage

Sport Macro Topic		
Keyword	**Absolute**	**Relative**
Select	879	99.65%
Ask	5	0.56%
Nested	43	4.87%
Distinct	605	68.59%
Limit	419	47.5%
Order by	317	35.94%
Filter	251	28.45%
Count	196	22.22%
Group by	138	15.64%
TOTAL	**882**	**100%**

(b) Sport keywords usage

Book Macro Topic		
Keyword	**Absolute**	**Relative**
Select	926	99.46%
Ask	7	0.75%
Nested	65	6.98%
Distinct	664	71.32%
Limit	384	41.24%
Order by	259	27.81%
Filter	299	32.11%
Count	168	18.04%
Group by	121	12.99%
TOTAL	**931**	**100%**

(c) Book keywords usage

Politics Macro Topic		
Keyword	**Absolute**	**Relative**
Select	756	99.6%
Ask	12	1.58%
Nested	69	9.09%
Distinct	521	68.64%
Limit	405	53.35%
Order by	304	40.05%
Filter	242	31.88%
Count	229	30.17%
Group by	169	22.26%
TOTAL	**759**	**100%**

(d) Politics keywords usage

Companies Macro Topic		
Keyword	**Absolute**	**Relative**
Select	705	99.15%
Ask	12	1.68%
Nested	86	12.09%
Distinct	507	71.3%
Limit	396	55.69%
Order by	371	52.18%
Filter	282	39.66%
Count	262	36.84%
Group by	259	36.42%
TOTAL	**711**	**100%**

(e) Companies keywords usage

Movie Macro Topic		
Keyword	**Absolute**	**Relative**
Select	985	99.89%
Ask	12	1.21%
Nested	110	11.15%
Distinct	676	68.55%
Limit	392	39.75%
Order by	405	41.07%
Filter	275	27.89%
Count	243	24.64%
Group by	212	21.5%
TOTAL	**986**	**100%**

(f) Movie keywords usage

Table 8.2: Keywords usage for each macro topic

9

Conclusions and Future Works

The data analysis carried out in this thesis was very stimulating and led me to develop new skills and new ways of dealing with the data available. The entire work of this thesis can be summarised in three main points:

- collect the data and analyze the SPARQL keywords usage
- create the ground truths of the Movie-related workflows
- use the ground truths to evaluate the students workflows

Initially, with all the Python notebooks available, I needed to be able to interact with them and find a way of extracting all the data, and that was my starting point. Once I realised that it was possible to load the notebooks into my script as a JSON files, everything was simplified by the great organisation of these files. The notebook cells became a simple array of values, much easier to handle.

The next step was to analyse all the queries to find the SPARQL keywords usage and to find a suitable structure to hold all this data so that it could be indexed according to the different entities involved: topics and students. I therefore created a 6-dimensional matrix, where each dimension corresponds to a specific category. In this way, it is possible to access this structure by selecting different parts of it as required, and thus generate statistics defined on a precise subset of the structure. The first part of my thesis relates to these first two phases and was a completely programming work.

The second main point, concerning the creation of ground truths, was basically to understand how entities and relationships are organised in Wikidata

given a certain workflow. At this stage, I was improving the knowledge I already had, because I had already done one of the workflows related to the films, and I was learning new knowledge about the other topics. Then, I focused on creating well-designed workflows that answered all questions as quickly and accurately as possible. Ultimately, given the need to use ground truths in the evaluation phase, I found a way to save their results so that they could be used by a script. This was perhaps the most challenging part, as the results needed to be stored in a clear and simple way so that they could be easily used later. It proved to be very difficult also because many of the tasks required were "exploratory", i.e. with open-ended questions with multiple answers.

Finally, the last part of the thesis was dedicated to the integration of the first two parts. Firstly, a script was created using ground truth files to evaluate the students' notebooks. Then, an output object was created containing all the information available for each notebook, both the use of SPARQL keywords and the evaluation. Finally this object is stored in a file in the student's folder. In the end, each student has a folder containing as many files as the number of workflows they have completed.

This thesis also analysed the different macro topics and the relationship between task requests and keyword usage. These analyses showed that, with a few exceptions, all queries used the `Select` construct. Furthermore, since Wikidata is very large and each entity can generally have hundreds of properties, the use of `Limit` was essential to truncate the result sets. Another important confirmation comes from the extensive use of the `Group By` keyword and aggregation functions, especially `Count`, as many tasks require aggregation. The use of `Limit` is so common that it was also widely used in `Group By` queries, along with sorting, to select and display only the best results.

Ground truths gave me the opportunity to explore Wikidata a lot, as on many occasions I had to compare the results of our Wikidata snapshot with the results of the online Wikidata. This made me realise the importance of reifications, as a lot of information is associated with a time interval. An example of this situation is shown in Figure 2.6: Douglas Adams was educated at two different schools, and his Wikidata page contains some qualifiers to add information about the time interval and the type of school. Many workflow tasks required examining changes over the years, and this was the most difficult type of task, as our Wikidata snapshot does not contain any reification.

Evaluation is very useful for checking basic tasks, but when the requirements

become more complex, people think differently. Obviously, when the tasks are general and it is not well defined what the answer should contain, the students try to do their best by answering according to their own logic. But even with more specific questions, where the query requires aggregation and a sub-query, there is always a phase of interpretation by the students and the results could be very different. For example: find the number of films in which Tarantino has appeared. One student might think of counting the distinct films and another might think of counting all the appearances, so that if Tarantino appeared in the same film as director and actor, that film would be counted twice. This is a simple example, but in general, with more complex requirements, these are the difficulties to be faced, and it is hard to evaluate such a workflow using only ground truth. This is because this kind of work is still exploratory search, and we must always remember the basics of exploratory search: you do not really know the domain and the schema, and you are never sure that you have found all the data and that it is correct. This is the greatest lesson of this thesis.

As far as future work is concerned, in agreement with the professors we plan to continue this research. In addition, new workflows with new topics were created in this second year of the Database 2 course. This expands our data set and allows us to refine the statistics and possibly confirm the conclusions drawn from this thesis. Looking at the workflow tasks available for the work on this thesis, we realised that the requirements were too general, so this year we have modified them to be more precise. This will make it easier to evaluate the workflows produced by the students against a ground truth. Remembering that we are always talking about exploratory search, it might seem that the choice to be so specific reduces the exploratory part of this work. In reality, there is rarely only one way to achieve a result, and Wikidata is an open knowledge base, so anyone can edit it. Even if there are standards to follow, one cannot be sure that people editing Wikidata entities are always using the correct standards. These errors are easily corrected for all known entities, as many people work on them, but in some cases the data remains incomplete or wrong.

Exploratory search is a really interesting area of research and, in fact, it is what we do every day by searching for information we do not know on search engines. This way of doing it just through querying a knowledge base allows us to learn a lot and understand the difficulties of managing so much interconnected data.

References

[1] World Wide Web Consortium. *Resource Description Framework (RDF) Model and Syntax Specification*. 1999. URL: https://www.w3.org/TR/PR-rdf-syntax/Overview.html.

[2] World Wide Web Consortium. *RDF 1.1 Turtle: Terse RDF Triple Language*. 2014. URL: https://www.w3.org/TR/turtle/#language-features.

[3] World Wide Web Consortium. *RDF 1.1 N-Triples: A line-based syntax for an RDF graph*. 2014. URL: https://www.w3.org/TR/n-triples/.

[4] World Wide Web Consortium. *RDF 1.1 XML Syntax*. 2014. URL: https://www.w3.org/TR/rdf-syntax-grammar/.

[5] Resa A. Roth Ryen W. White. *Exploratory Search: Beyond the Query-Response Paradigm*. 2009.

[6] Gary Marchionini. "Exploratory search: from finding to understanding". In: *Communications of the ACM* 49.4 (2006), pp. 41–46.

[7] *DBpedia*. URL: https://www.dbpedia.org.

[8] *Welcome to Wikidata*. URL: https://www.wikidata.org/wiki/Wikidata:Main_Page.

[9] Wikimedia Foundation Inc. *Wikimedia Foundation*. URL: https://en.wikipedia.org/wiki/Wikimedia_Foundation.

[10] Wikimedia Foundation Inc. *Wikimedia Foundation Mission*. URL: https://wikimediafoundation.org/about/mission/.

[11] *Wikidata Query Service*. URL: https://query.wikidata.org.

[12] World Wide Web Consortium. URL: https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/.

[13] OpenLink Software. URL: https://virtuoso.openlinksw.com.

[14] OpenLink Software. URL: https://docs.openlinksw.com/virtuoso/logicaldatamodel/.

[15] OpenLink Software. URL: https://docs.openlinksw.com/virtuoso/colstore/.

[16] WDQS Search Team. "WDQS Backend Alternatives". In: (2022).

[17] *NumPy*. URL: https://numpy.org.

Acknowledgments

I would like to express my deepest appreciation to the University of Padova and all the professors I have met over the years, who have always been well prepared in their teaching. In particular to the Professors Gianmaria Silvello and Matteo Lissandrini (Aalborg University), who supervised me in the research training and in the Master Thesis. They gave me the opportunity to study and work on a topic of great importance.

I would especially like to thank my family who gave special support to my career by always believing in me and my academic path when things did not always go well. I encountered several difficulties but it is thanks to my parents that I can celebrate this success today. The rest of my family, my aunts, uncles and grandparents were also important in achieving this goal, reassuring me day after day about my capabilities and potential to achieve my goals.