Jaime Domingos Marques

AUTOMATIC DATA MODEL CONVERSION

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering advised by Professor Ph.D. Catarina Helena Branco Simões da Silva and Engineer João Garcia and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2022

This page is intentionally left blank.

Faculty of Sciences and Technology

Department of Informatics Engineering

Automatic Data Model Conversion

Jaime Domingos Marques

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised by Professor Ph.D. Catarina Helena Branco Simões da Silva and Engineer João Garcia and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2022

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Acknowledgements

I would like to express my sincere gratitude to everyone who, directly or indirectly, contributed to the elaboration of this dissertation. Whether it is for your concrete tips or simple encouragement words, thank you!

This page is intentionally left blank.

Abstract

As urban areas grow and become more developed, municipal governments look for new ways of managing their cities. Urban Platform is Ubiwhere's product which aims at fulfilling this necessity, presenting interactive maps and statistics in real time that help the decision process.

However, as new cities adhere to the platform, the number of data sources also increases and their manual integration in the Urban Platform becomes time-consuming and labour-intensive.

The present dissertation contributes to tackle this problem, addressing the integration of new data models into the Urban Platform as an automatic schema matching task.

First, a state of the art of the techniques and systems currently used is provided. Among them are natural language processing, graph theory and matrix combination techniques. Then, all the steps for building the automatic solution are detailed.

The proposed solution includes four main steps: the source handling and entity recognition; the selection of candidate pairs of entities from the Urban Platform and from the new data source; the similarity calculation of those pairs; the extraction of a final mapping between the new source and the Urban Platform.

The experimental study showed the solution's potential. Yet it is far from being production-ready, since the average f1-score (61%) is still not high enough to be worth replacing the traditional manual integration, particularly regarding harder matching cases.

Keywords

Data Integration, Schema Matching, Schemas, Ontologies, Natural Language Processing, Smart Cities, Urban Platform

This page is intentionally left blank.

Resumo

Com o crescente desenvolvimento das áreas urbanas, os governos municipais procuram novas formas de gerirem as suas cidades. A *Urban Platform* é o produto da empresa Ubiwhere que procura dar resposta a este problema, apresentando mapas interativos e estatísticas em tempo real, que ajudam na tomada de decisões.

Porém, com o aumento do número de novas cidades a aderirem à plataforma, o número de fontes de dados a serem integradas na plataforma também aumenta e a sua integração manual na *Urban Platform*, torna-se muito morosa e dispendiosa no que diz respeito à mão-de-obra.

A presente dissertação visa contribuir para a resolução desta problemática, abordando esta integração de novos modelos de dados na *Urban Platform* como um problema de correspondência automática entre esquemas.

Primeiramente, é apresentado o estado da arte das técnicas e sistemas atualmente utilizados nesta área. Entre elas estão técnicas de processamento de linguagem natural, de teoria de grafos e técnicas de combinação matricial. Posteriormente, são detalhados todos os passos para o desenvolvimento do programa.

A solução proposta é constituída por quatro etapas principais, sendo elas: a leitura das fontes e reconhecimento de entidades; a seleção de pares candidatos constituídos por entidades do modelo de dados da *Urban Platform* e do modelo de dados da nova fonte; o cálculo das similaridades desses pares; a extração de mapeamentos finais entre o modelo da nova fonte e o modelo da *Urban Platform*.

As experiências realizadas demonstraram algum potencial desta solução. Contudo, esta ainda se encontra longe de poder ser colocada em produção, pois o valor médio do f1-score (61%) ainda não é suficientemente elevado para que compense substituir a correspondência tradicional, sobretudo no que toca aos casos de maior dificuldade de correspondência.

Palavras-Chave

Integração de Dados, Correspondência entre Esquemas, Esquemas, Ontologias, Processamento de Linguagem Natural, Cidades Inteligentes, Urban Platform

This page is intentionally left blank.

Acronyms

AHP Analytic Hierarchy Process.

AOA Attention-over-Attention.

APFEL Alignment Process Feature Estimation and Learning.

API Application Programming Interface.

BERT Bidirectional Encoder Representations from Transformers.

BiLSTM Bidirectional Long Short Term Memory.

BMPM Beider-Morse Phonetic Matching.

BPE Byte-Pair Encoding.

CBoW Continuous Bag of Words.

CBSR Controlled Batch Sample Ratio.

COMA Combination of Matching Algorithms.

CPU Central Processing Unit.

CSV Comma-separated Values.

DAG Directed Acyclic Graph.

DC Dublin Core.

DOLCE Descriptive Ontology for Linguistic and Cognitive Engineering.

ELMo Embedding from Language Models.

FOAF Friend of a Friend.

GIL Global Interpreter Lock.

GloVe Global Vectors.

GUI Graphical Interface.

HTTP Hypertext Transfer Protocol.

IRI Internationalized Resource Identifier.

JSON JavaScript Object Notation.

JSON-LD JavaScript Object Notation for Linked Data.

KSTEM Krovetz Stemmer.

LCA Lowest Common Ancestor.

LCS Longest Common Sub-string.

LEAPME LEArning-based Property Matching with Embeddings.

LOD Linked Open Data.

LSD Learning Source Descriptions.

LSTM Long Short-Term Memory.

N3 Notation 3.

NGD Normalized Google Distance.

NLP Natural Language Processing.

NLTK Natural Language Toolkit.

OAEI Ontology Alignment Evaluation Initiative.

OID Object Identifier.

OWA Ordered Weighted Averaging.

OWL Ontology Web Language.

POSIX Portable Operating System Interface.

RAM Random Access Memory.

RDF Resource Description Framework.

RDFS Resource Description Framework Schema.

REST Representational State Transfer.

RF4SM Random Forest for Schema Matching.

SAX Simple API for XML.

SDG Sustainable Development Goal.

SemInt Semantic Integrator.

SMB Schema Matcher Boosting.

SPARQL SPARQL Protocol and RDF Query Language.

SQL Structured Query Language.

SUMO Suggested Upper Merged Ontology.

ToS Threshold of Success.

UBP Urban Platform.

URI Uniform Resource Identifier.

W3C World Wide Web Consortium.

XML Extensible Markup Language.

XSD XML Schema Definition.

YAGO Yet Another Good Ontology.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

According to the 2018 United Nations' Prospects on World Urbanization, more than 55% of the world's population lives in cities and that value is expected to shoot up to 68% until 2050 [24].

Modern cities are much more than concrete jungles. They are living entities that grow, each one at its own pace, having their population as the blood which fills and feeds the entire organism. Like any other living thing, a city needs a brain. This role is played by municipal governments who control the operations and call the shots when something fails.

As modern cities turn into mega cities, their management becomes harder, since many events occur at the same time. Luckily for city mayors, technology can facilitate that.

"Smart cities" was a term first used in the 1990's [153], although the concept emerged in the 1970's with the Los Angeles' "Community Analysis Bureau" [7] where collected data was analysed to establish city policies.

The concept evolved to something much more than just that, so nowadays we have platforms like Ubiwhere's Urban Platform (UBP)[1] on which relevant data is displayed over interactive maps. These platforms auxiliate city administrative teams to visualize all occurrences [22]. At the same time, they show real-time statistics, allowing a quick and cross-domain analysis of city data. But how can we generalize a single platform to work properly for every city? Different cities have different inputs producing all sorts of data structures and formats. How can we read, interpret and display these data correctly on a platform?

This dissertation aims at finding an automatic solution to standardize the data structures arriving at the Urban Platform.

1.1 Context and Motivation

UBP is a city management all-in-one tool, launched in 2018 by Ubiwhere, which shows a variety of different figures, such as traffic, parking occupancy and air quality in a centralized dashboard [22]. The platform's targets are teams of people working for municipal services, whether they are infrastructure personnel or even decision makers who need to analyse real-time data in order to make the best decision possible (Figure 1.1). The control panel offers a map where the occurrences are displayed in a more pleasant way (Figure 1.2).

[1] https://urbanplatform.city

Urban Platform's versatility allows clients to customize the panel and stack different layers of information. The map is able to host new information collected by different sources, from sensors to cameras.

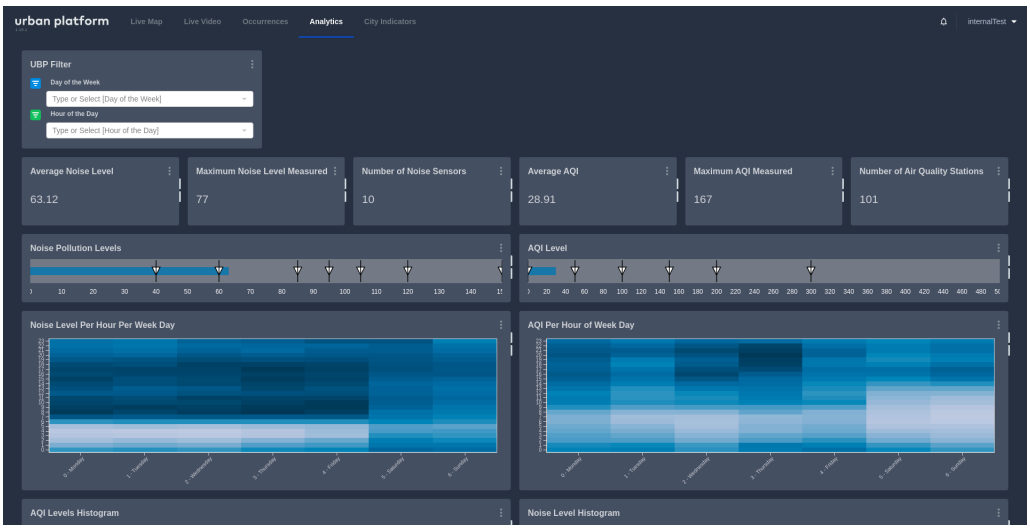

Figure 1.1: Analytics on Urban Platform

Figure 1.2: Urban Platform's Live Map

Besides the scope of features enumerated before, the potential of this platform goes way beyond that. Sustainability issues often arise when the subject is big cities. Projects like the UBP are important for responsible resource management. For instance, if a water pipe leaks water and the incident is reported on the platform, it allows authorities to act quicker avoiding the wastage of thousands of liters. In addition, it may be useful to monitor specific parameters, like air quality, important for public health. All things combined contribute for the accomplishment of the Sustainable Development Goals (SDGs) [25] in matters like "Sustainable Cities and Communities", "Good Health and Well Being", "Industry, Innovation and Infrastructure" and "Responsible Consumption and Production" [5]. Details about Urban Platform's architecture are going to be presented later on.

UBP needs to stay modular as new cities adopt the platform and features are added. New cities mean new sources of data, which can introduce heterogeneity, from the file format (e.g. JSON, XML, CSV...) to the way the data is organized inside the file. When the files

reach UBP, they are processed. The information is extracted and put into a standardized form. In other words, there is a matching between the attributes on both ends. In the literature this process is often called schema matching [85] and it is part of a wider data integration task. This case study has the particularity of one of the end-points being a semi-structured data file and the other one a rigid relational schema to which we want to correspond the first one. This poses several challenges such as the amount of instance data to work with, since it will depend on how filled is the database.

In the current UBP's implementation, the matching between attributes is done manually. Developers inspect the files, make decisions and cover all the scenarios resorting to traditional conditions. We may call it not practical, slow or boring, but it works. However, if a huge number of cities decide to adopt the platform, having the sensing devices their own exporting formats, the reliance on human intervention becomes unbearable.

The motivation for this project comes from the possibility of creating a tool that performs the matching automatically, facilitating the integration and reducing human interaction and effort, as much as possible. By automatically corresponding arriving data to the right models on the system, the flexibility of the platform increases, since there is no need for installing different sensors from the ones that already exist, thus significantly lowering UBP's installation costs for new cities.

1.2 Challenges

Since it is a complex task, we may divide the main problem into several smaller yet still complex challenges.

File handling: This first step is responsible for reading the file content. The files come in many different formats. The program has to take all, or at least the most used ones into consideration.

Recognize the different entities: After reading the file, one needs to recognize the structures and entity hierarchy present in the file.

Perform the similarity evaluation: Probably the biggest challenge of them all is to extract the similar entity pairs. Several sub-problems might appear depending on the file. Some of those are listed as follows:

> **Fuzzy Matching:** In certain cases, it is not possible to get a direct match, for instance if an attribute provides us more information than the one we need to characterize the destination entity. In these scenarios we can match entities based on part of the information [35]. Ultimately we want to do schema or model matching which is a mapping between objects that share semantic relationships. Such process is hampered by representational (file extension), structural (different attribute hierarchy and data types), syntactic (different idioms can be present, for instance) and semantic (since the same entity may be referred via different terms) differences [135][28].
>
> **Entity Merging and Merge-Purge/Record Deduplication:** Entity merging is necessary when the target entity represents a broader concept, where two or more source entities fit. Finally, when mapping the entities, the records need to be merged, a process called merge-purge or record deduplication, in order to get a unique piece of information that meets the destination entity's requirements and to avoid redundancy [35][54].

Validate the solution: Finally, we need to find a way to validate the solution found. This includes building or finding a testing dataset and designing the experiment so that useful conclusions can be drawn and the actual requirements to be met are correctly tested.

1.3 Goals

The main goal is to develop an automatic data model conversion system which overcomes the challenges stated above. In other words, a program that is able to handle sources of different formats, recognize their structure and entities, determine similarities between entities of different sources, and calculate statistics that contribute to its validation.

Its components must be modular, so that future additions can be done effortlessly by developers who have never had contact with the project.

This system must work at least for the source files found to be the most heterogeneous or for the UBP's models used in most cities, such as traffic readings, air quality or parking occupancy.

Ideally, the solution should outperform the manual integration currently in place, especially regarding execution times.

1.4 Document Structure

This document acts in accordance to the following structure. The second chapter is where basic background concepts and formal definitions are presented for better contextualization and understanding of the problem.

The third chapter corresponds to the State of the Art, where a combination of strategies and attempts of solutions are reported and analyzed.

The fourth chapter is reserved for planning. It is discussed how the project was be conducted, as well as the different tasks to be completed and the risks associated with them.

The fifth chapter of this document is dedicated to present specific attributes the new component should have and the architectural choices that materialize them.

The sixth chapter explains how the different components of the proposed solution were implemented.

The seventh chapter is a description of the testing phase. This includes what and how experiments were done, and their results.

The last chapter, the eighth, concludes the dissertation and gives some hints about what can be done in the future to improve the current solution.

Chapter 2

Background

This chapter provides an understanding of the background fields explored when writing this dissertation. Even though some may not be directly related to the project's main theme, it is important to address these subjects to understand the problem in its full extension and making rigorous decisions when planning a solution.

The first section is dedicated to break down into details the Urban Platform's architecture, as well as locating the system components belonging to the project's area of interest. This section is followed by some more theoretical ones, being them in the order in which they appear the "Semantic Web", where ontologies and schemas are introduced, and "Data Integration and Sub-Problems", dedicated to formally characterize schema matching and adjacent topics. The chapter finalizes with a conclusion and sum up of the concepts discussed.

2.1 Urban Platform Architecture

The Urban Platform (UBP) was launched in 2018 as Ubiwhere's urban management assistant tool. The platform can be deployed for any city and it shows gathered data from sensors that can either be mounted by Ubiwhere or reused from previous applications. The challenge has been expanding UBP to new cities without worrying too much about different sensor exporting formats while doing the bare minimum adjustments to the code.

Figure 2.1 represents the architecture of UBP's current implementation. The data flow follows three main stages: import, processing and persistence.

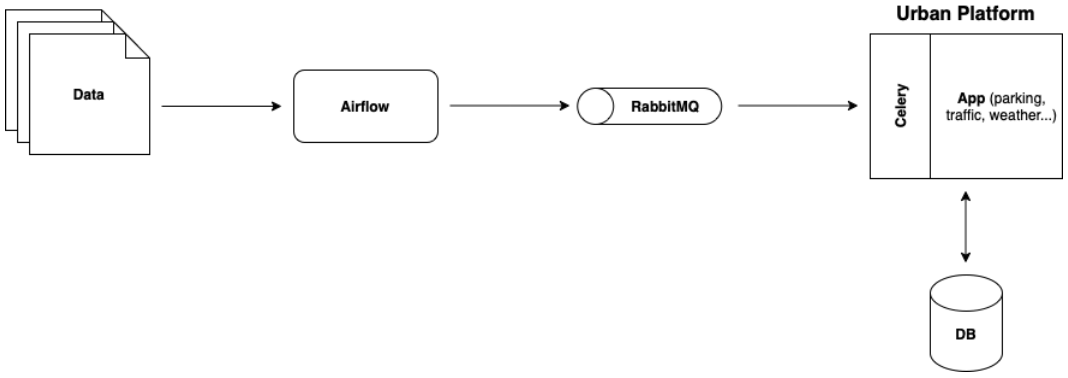

Figure 2.1: UBP Architecture

The files are primarily imported to Airflow. Apache Airflow[1] is an open-source workflow manager prepared to deal with very complex orchestrations. Its main features are scheduling and automation but it also helps in monitoring and diagnosing issues in workflows [1][2]. Data pipelines are represented as Directed Acyclic Graph (DAG) which are encoded in python files. There are several instances of the UBP, one for each city, having them their own DAGs according to what needs to be imported. Each DAG has the task of importing and sending a file corresponding to one specific type of statistics (parking, weather, air quality etc.). The imported file is formatted and sent to Rabbit Message Queue, a popular open-source message broker which implements Advanced Message Queuing Protocol (AMQP). RabbitMQ[2] receives messages and sends them to subscribers, in this case, UBP's Celery[3], a library to run asynchronous processes [4]. The imported files are persisted at UBP's database for later use. Then, tasks are distributed across different Django apps, so that data can be requested and displayed in front-end. These apps have their separate functionalities (e.g. parking, traffic, air quality data) and their use is optional, depending on what is meant to be shown on the platform. These characteristics contribute for the modularity of UBP, making it suitable for most city scenarios.

Having all those steps in mind, it is safe to say that the main concern for this research is the processing stage, more specifically inside the Airflow, because it is where most code changes happen when creating new UBP instances. On behalf of the platform's scalability, a new correspondence system will have to be designed so that the data parsing occurs as if all data is consistently coming from the same source.

2.2 Semantic Web

Semantic Web is an extension of the World Wide Web that attempts to define content elaborated by humans in a way that it is meaningful to machines [16][17]. As Sir Tim Berners-Lee described in [146][40], semantic web brings structure to content, which is then parsed by the computer and displayed into a human readable form.

Semantic Web is therefore the basis of data integration. Anything, whether it is commercial, medical, or academic information, can be put online and different applications are able to share files with each other.

This new dimension is achieved by several standards which are established by the World Wide Web Consortium (W3C)[4]. They promote common formats and protocols around the web and different models and languages are suited for different purposes [18]. Having that in mind, we can present the semantic web as a stack of different building blocks (Figure 2.2), each one with its function [79].

The foundation of the stack is the web platform already in existence. It includes well-established standards like Uniform Resource Identifier (URI)[5] or Internationalized Resource Identifier (IRI)[6], an international version of URIs, to uniquely identify the data, Unicode[7] to encode the message and protocols like Hypertext Transfer Protocol (HTTP)[8] for communication.

[1] https://airflow.apache.org
[2] https://www.rabbitmq.com
[3] https://docs.celeryq.dev/en/stable/
[4] https://www.w3.org/Consortium/
[5] https://www.w3.org/wiki/URI
[6] https://www.ietf.org/rfc/rfc3987.txt
[7] https://datatracker.ietf.org/doc/html/rfc5198
[8] https://datatracker.ietf.org/doc/html/rfc2616

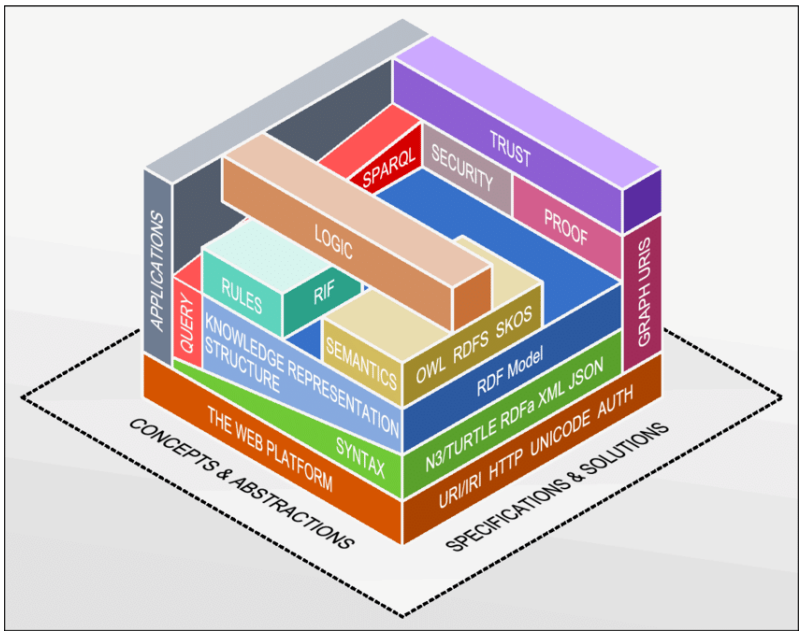

Figure 2.2: The Semantic Web stack [15]

The second layer refers to the document types. The formats provide the syntax for the document. Among these serialization formats are Extensible Markup Language (XML), Turtle, N-Triples and JavaScript Object Notation for Linked Data (JSON-LD).

The information interchange block is above the format one. This layer represents the data model, the content of the file. Considering the analogy of a book [9], the book can be in braille, digital or in regular print. Despite not everyone being able to read the book in braille, for example, all those formats represent the same information.

The standard used is Resource Description Framework (RDF), which connects resources of different types via uniform triples: subject, predicate and object [26].

Bellow, an example is presented [23] where the same linked data is represented in different formats.

Turtle: Turtle is a RDF syntax with file extension .ttl, compatible with Notation 3 (N3), that allows the files to be written in a compact and human friendly way (Figure 2.3) [20]. The URIs are enclosed in <> similarly to other syntaxes. To get a much cleaner definition, URI abbreviations are possible. Such abbreviations are defined as prefixes (`@prefix`) so that the same URI is not repeated over and over. Predicates may be abbreviated as well, which is convenient to keep the reading simple. E.g. the URI `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` is equivalent to the keyword "a".

N-Triples: N-Triples is a common format used to store and read RDF since the triples can easily form a direct knowledge graph. The file extension is .nt and the syntax is as simple as (<subject> <predicate> <object>). The linked data may be obtained using knowledge bases that extract information from websites such as Wikipedia[9], WordNet[10], GeoNames[11]. Some examples are DBpedia [93], Wikidata [149] and YAGO [120]. Objects can be URIs or simply strings, called literals,

[9]https://www.wikipedia.org
[10]https://wordnet.princeton.edu/
[11]https://www.geonames.org

```
@prefix dbr:    <http://dbpedia.org/resource/> .
@prefix dbo:    <http://dbpedia.org/ontology/> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix geo:    <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .
@prefix schema: <http://schema.org/> .

dbr:Bob_Marley
   a foaf:Person ;
   rdfs:label "Bob Marley"@en ;
   rdfs:label "Bob Marley"@fr ;
   rdfs:seeAlso dbr:Rastafari ;
   dbo:birthPlace dbr:Jamaica .

dbr:Jamaica
   a schema:Country ;
   rdfs:label "Jamaica"@en ;
   rdfs:label "Giamaica"@it ;
   geo:lat "17.9833"^^xsd:float ;
   geo:long "-76.8"^^xsd:float ;
   foaf:homepage <http://jis.gov.jm/> .
```

Figure 2.3: Turtle Syntax [23]

to which we can attach tags, marked by @ or even constraint the type of an object using ^^ (Figure 2.4). The latter is useful to ensure the validity of an assigned object. Ontologies (e.g. Friend of a Friend (FOAF), Good Relations, Dublin Core (DC)) [6] can also be used to connect different entities.

```
<http://dbpedia.org/resource/Bob_Marley> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
<http://dbpedia.org/resource/Bob_Marley> <http://www.w3.org/2000/01/rdf-schema#label> "Bob Marley"@en .
<http://dbpedia.org/resource/Bob_Marley> <http://www.w3.org/2000/01/rdf-schema#label> "Bob Marley"@fr .
<http://dbpedia.org/resource/Bob_Marley> <http://www.w3.org/2000/01/rdf-schema#seeAlso> <http://dbpedia.org/resource/Rastafari> .
<http://dbpedia.org/resource/Bob_Marley> <http://dbpedia.org/ontology/birthPlace> <http://dbpedia.org/resource/Jamaica> .
<http://dbpedia.org/resource/Jamaica> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://schema.org/Country> .
<http://dbpedia.org/resource/Jamaica> <http://www.w3.org/2000/01/rdf-schema#label> "Jamaica"@en .
<http://dbpedia.org/resource/Jamaica> <http://www.w3.org/2000/01/rdf-schema#label> "Giamaica"@it .
<http://dbpedia.org/resource/Jamaica> <http://www.w3.org/2003/01/geo/wgs84_pos#lat> "17.9833"^^<http://www.w3.org/2001/XMLSchema#float> .
<http://dbpedia.org/resource/Jamaica> <http://www.w3.org/2003/01/geo/wgs84_pos#long> "-76.8"^^<http://www.w3.org/2001/XMLSchema#float> .
<http://dbpedia.org/resource/Jamaica> <http://xmlns.com/foaf/0.1/homepage> <http://jis.gov.jm/> .
```

Figure 2.4: N-Triples [23]

RDF/XML: RDF/XML is the oldest format. It has some features similar to turtle, such has prefixes, which help the file to be a little more intelligible (Figure 2.5). It is losing popularity, although it is still considered a standard format.

JSON-LD: JSON-LD is harder to interpret by humans comparing to the other formats (Figure 2.6). Despite the decreasing in popularity, it has the advantage of using an already existent format (JSON) [14], just like RDF/XML. Entities are defined at the top of hierarchical blocks while data types and literals stay within the blocks.

The fourth block has the task of building the classes, relations and restrictions between them. They are expressed by languages such as Resource Description Framework Schema (RDFS) and Ontology Web Language (OWL), the highest level of expressivity. The definition of such relationships is supported by rules, logics and proofs. Here an example in Turtle adapted from [27][21][13] shows the usage of RDF, RDFS and OWL (Listings 2.1, 2.2 and 2.3). By comparing side by side, the differences in expressivity are notorious (Figure 2.7).

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:ns0="http://dbpedia.org/ontology/"
         xmlns:schema="http://schema.org/"
         xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">

  <foaf:Person rdf:about="http://dbpedia.org/resource/Bob_Marley">
    <rdfs:label xml:lang="en">Bob Marley</rdfs:label>
    <rdfs:label xml:lang="fr">Bob Marley</rdfs:label>
    <rdfs:seeAlso rdf:resource="http://dbpedia.org/resource/Rastafari"/>
    <ns0:birthPlace>
      <schema:Country rdf:about="http://dbpedia.org/resource/Jamaica">
        <rdfs:label xml:lang="en">Jamaica</rdfs:label>
        <rdfs:label xml:lang="it">Giamaica</rdfs:label>
        <geo:lat rdf:datatype="http://www.w3.org/2001/XMLSchema#float">17.9833</geo:lat>
        <geo:long rdf:datatype="http://www.w3.org/2001/XMLSchema#float">-76.8</geo:long>
        <foaf:homepage rdf:resource="http://jis.gov.jm/"/>
      </schema:Country>
    </ns0:birthPlace>

  </foaf:Person>

</rdf:RDF>
```

Figure 2.5: RDF/XML [23]

```
[
  {
    "@id": "http://dbpedia.org/resource/Bob_Marley",
    "@type": ["http://xmlns.com/foaf/0.1/Person"],
    "http://www.w3.org/2000/01/rdf-schema#label": [
      {"@value": "Bob Marley", "@language": "en"},
      {"@value": "Bob Marley", "@language": "fr"}
    ],
    "http://www.w3.org/2000/01/rdf-schema#seeAlso": [{"@id": "http://dbpedia.org/resource/Rastafari"}],
    "http://dbpedia.org/ontology/birthPlace": [{"@id": "http://dbpedia.org/resource/Jamaica"}]
  },
  {
    "@id": "http://dbpedia.org/resource/Jamaica",
    "@type": ["http://schema.org/Country"],
    "http://www.w3.org/2000/01/rdf-schema#label": [
      {"@value": "Jamaica", "@language": "en"},
      {"@value": "Giamaica", "@language": "it"}
    ],
    "http://www.w3.org/2003/01/geo/wgs84_pos#lat": [
      {"@value": "17.9833", "@type": "http://www.w3.org/2001/XMLSchema#float"}
    ],
    "http://www.w3.org/2003/01/geo/wgs84_pos#long": [
      {"@value": "-76.8", "@type": "http://www.w3.org/2001/XMLSchema#float"}
    ],
    "http://xmlns.com/foaf/0.1/homepage": [{"@id": "http://jis.gov.jm/"}]
  },
  {"@id": "http://dbpedia.org/resource/Rastafari"},
  {"@id": "http://jis.gov.jm/"},
  {"@id": "http://schema.org/Country"},
  {"@id": "http://xmlns.com/foaf/0.1/Person"}
]
```

Figure 2.6: JSON-LD [23]

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

:John    rdf:type   :Man .
:John    :livesIn "New-York" .
:livesIn     rdf:type     rdf:Property .
```

Listing 2.1: RDF example

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
...

:John    rdf:type   :Man .
:Man     rdfs:subClassOf   :Human .
:John    :livesIn "New-York" .
:livesIn     rdf:type     rdf:Property .
```

Listing 2.2: RDFS example

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
...

:livesIn     rdf:type     owl:DatatypeProperty .
:Human   rdf:type     owl:Class .
:Man     rdf:type     owl:Class .
:Man     rdfs:subClassOf   :Human .
:John    rdf:type   :Man .
:John    rdf:type     owl:NamedIndividual .
```

Listing 2.3: OWL example

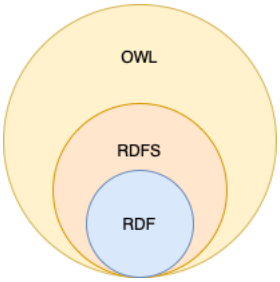

Figure 2.7: Expressivity of RDF, RDFS and OWL in a diagram form

The last two layers combined add semantics to data, allowing the creation of a knowledge graph. A knowledge graph is an interlinked, interoperable and flexible information structure. Different elements of a data structure have now a meaning. Figure 2.8 is the knowledge graph for the Bob Marley's example presented before. Circles represent entities, rectangles literals and arrows the predicates.

Knowledge graphs are an excellent framework for data integration, since they combine expressivity, because various types of data can be represented, formal and standardized definition of semantics, which allows both humans and machines worldwide to interpret the graph and interoperability, backed up by SPARQL Protocol and RDF Query Language (SPARQL), a language similar to SQL used to query over knowledge graphs. All this comes with the possibility of efficient graph management [26].

2.2.1 Schemas and Ontologies

In the last section the different layers of the semantic web were characterized. This section rather focuses on formally define schemas and ontologies that are used in the matching process. A schema is defined as a set of elements linked by a structure [76]. XML Schema Definition (XSD) is perhaps the most well-known Schema language out there, used to express rules and specify data types a XML file must conform to.

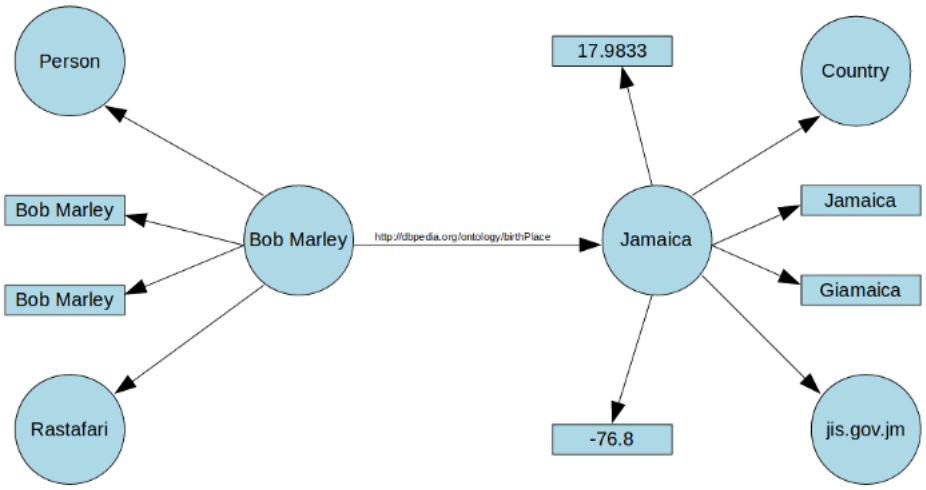

Figure 2.8: Knowledge graph of the Bob Marley's example [23]

Definition 1. Formally, a schema S is defined by $S = (I^S, Lab^S, F^S, R^S)$ [156] where:

1. $I^S = \{s_1, s_2, ..., s_n\}$, $I^S \neq \emptyset, n \in \mathbb{N}, n < \infty$. Each idividual s_i $(i = \{1, 2, ..., n\})$ is uniquely identified by an Object Identifier (OID). The size of this set is the size of the schema $|I^S| = |S| = n$;
2. $Lab^S = \{Lab_1^S, Lab_2^S, ..., Lab_i^S\}$, $i \in \mathbb{N}$, $i < \infty$ is constant. Each Lab_i^S is a string describing an individual n;
3. $F^S = \{f_1^S, f_2^S, ..., f_j^S\}$, $j \in \mathbb{N}$, $j < \infty$ a set of labeling functions for each individual s_n $(i = \{1, 2, ..., n\})$;
4. $R^S = \{R_1, R_2, ..., R_k\}$, $R^S \neq \emptyset, k \in \mathbb{N}, k < \infty$ is the set of relations between individuals. In general the relations are binary, but if the relations are b-ary, then $R \subseteq (I^S)^b$.

In practical example adapted from [155] the schema definition can be seen in action.

```
<schema xmlns="...">
    <complexType name="Product">
        <element name="ProductID" type="xs:int"/>
        <element name="ProductName" type="xs:string"/>
        <element name="ProductType" type="xs:string"/>
    </complexType>
</schema>
```

After a short analysis, it is deductible that $I^S = \{s_1, s_2, s_3, s_4\}$ and that $|S| = |I^S| = 4$. The label collection is composed by the name set $N^S = \{Product, ProductID, ProductName, ProductType\}$, the concept set of individuals $C^S = \{(product\#n\#1), (ID\#n\#2), (name\#n\#1), (type\#n\#1)\}$ obtained by an external source such as WordNet, two type sets for individuals $T_1^S = \{complexType, element\}$ and $T_2^S = \{xs : int, xs : string\}$ and finally a set for relations $L_R^S = \{include\}$ (in this example inclusion is the only relation present in the schema). So, $Lab^S = \{N^S, C^S, T_1^S, T_2^S, L_R^S\}$. From that we get five different label functions $F^S = \{f_1, f_2, f_3, f_4, f_5\}$. f_1 is the funtion to attribute the name set of

individuals (e.g. $f_1(s_1) = Product, f_1(s_2) = ProductID...$), f_2 for obtaining the concept set of individuals and so on and so forth. It is important to note the fifth label function has two inputs, because the relations require more than one individual (e.g. $f_5(s_1, s_2) = include$). Given the simplicity of this case study, the only relations are the ones with individual s_1: $R^S = \{(s_1, s_2), (s_1, s_3), (s_1, s_4)\}$. Since the schema has several labels, it is called a multi-labeled schema. These labels provide crucial information for the matching process.

In Philosophy, Ontology is a sub-field of Metaphysics, which deals with the study of "what exists" [147][139]. When applied to Computer Science, an ontology is the abstract model of shared concepts [73][116][44]. In other words, it is the formal representation of the properties and relations between different entities of a domain.

Definition 2. Being O an ontology, $O =< C, I, R, P, A >$, where C is a set of classes, I the set of all individuals, R the set of relations, P the set of data types and A the set of axioms [74]. Other equivalent definitions can be found in [84][65]

An ontology describes a vocabulary, which contains terms, relationships between those terms and axioms on using the terms [11].

RDFS and OWL are both languages for writing ontologies. Despite being in the same layer of the semantic web stack, RDFS are more commonly used to describe simpler vocabularies than OWL, a much more powerful language, although it is more complex to implement. Behind OWL there is a description logic called SROIQ$^{(D)}$ (Figure 2.9 2.10). Thus, OWL can also be represented by description logic:

```
...
    :publicationYear a owl:DatatypeProperty;
            rdfs:domain :Book;
            rdfs:range xsd:integer.
```

It is equivalent to $\exists\ publicationYear.\top \sqsubseteq Book$, $\top \sqsubseteq \forall\ publicationYear.Integer$ [127]

	Syntax	Semantics
Individuals:		
individual name	a	a^I
Roles:		
atomic role	R	R^I
inverse role	R^-	$\{\langle x, y\rangle \mid \langle y, x\rangle \in R^I\}$
universal role	U	$\Delta^I \times \Delta^I$
Concepts:		
atomic concept	A	A^I
intersection	$C \sqcap D$	$C^I \cap D^I$
union	$C \sqcup D$	$C^I \cup D^I$
complement	$\neg C$	$\Delta^I \setminus C^I$
top concept	$\top$	Δ^I
bottom concept	$\bot$	$\emptyset$
existential restriction	$\exists R.C$	$\{x \mid \text{some } R^I\text{-successor of } x \text{ is in } C^I\}$
universal restriction	$\forall R.C$	$\{x \mid \text{all } R^I\text{-successors of } x \text{ are in } C^I\}$
at-least restriction	$\geqslant n\ R.C$	$\{x \mid \text{at least } n\ R^I\text{-successors of } x \text{ are in } C^I\}$
at-most restriction	$\leqslant n\ R.C$	$\{x \mid \text{at most } n\ R^I\text{-successors of } x \text{ are in } C^I\}$
local reflexivity	$\exists R.Self$	$\{x \mid \langle x, x\rangle \in R^I\}$
nominal	$\{a\}$	$\{a^I\}$
where $a, b \in \mathsf{N}_I$ are individual names, $A \in \mathsf{N}_C$ is a concept name, $C, D \in \mathbf{C}$ are concepts, $R \in \mathbf{R}$ is a role		

Figure 2.9: SROIQ$^{(D)}$ [91]

	Syntax	Semantics
ABox:		
concept assertion	$C(a)$	$a^I \in C^I$
role assertion	$R(a,b)$	$\langle a^I, b^I \rangle \in R^I$
individual equality	$a \approx b$	$a^I = b^I$
individual inequality	$a \not\approx b$	$a^I \neq b^I$
TBox:		
concept inclusion	$C \sqsubseteq D$	$C^I \subseteq D^I$
concept equivalence	$C \equiv D$	$C^I = D^I$
RBox:		
role inclusion	$R \sqsubseteq S$	$R^I \subseteq S^I$
role equivalence	$R \equiv S$	$R^I = S^I$
complex role inclusion	$R_1 \circ R_2 \sqsubseteq S$	$R_1^I \circ R_2^I \subseteq S^I$
role disjointness	$Disjoint(R,S)$	$R^I \cap S^I = \emptyset$

Figure 2.10: SROIQ$^{(D)}$ [91]

All Semantic web technologies described in this section are somehow related and are meant to be inter-operable, that is why, sometimes, these concepts may seem hard to properly specify. However, all sources seem to be consensual on the fact that schemas are more focused on data and its structure, whereas ontologies are more focused on semantics. The choice on one or the other depends on the complexity of the model we are dealing with.

2.3 Data Integration and Sub-Problems

Semantic Web was meant to provide an extra layer of information to the contents of the World Wide Web, so websites could interpret and derive new knowledge based on that. However, the Semantic Web did not reach the scale initially envisioned. The languages are complex, the metadata has to be updated constantly to avoid becoming obsolete, and the interoperability between different sites or institutions was not facilitated enough. Therefore, the goal of enriching all the web content with a new layer of metadata, was not fulfilled entirely. It is fair to say that it left its mark, though. The developed technologies survived and keep being used successfully, mostly on a smaller scale, in companies or archives, for instance [70].

Data integration is still a problem and, depending on the application domain, it may be expressed in many different sub-problems [35][60]. For instance, in the medical domain, hospitals may need to merge patient records so that a single registry has all the information from one patient. In addition to that, in distributed systems, database applications may differ and, in order to integrate them, a matching between database schemas is required. When interoperability between heterogeneous sources is required the problem of ambiguous interpretation of concepts arises. Hence, corporations face an increasingly harder challenge, because the linkage is rarely a straightforward procedure. Heterogeneity in schemas or ontologies is organized in different levels [65]:

Syntactic heterogeneity: It occurs when the two sources are not written in the same language. The challenge here is to convert the schemas or ontologies to be matched in a standard format, while preserving all the knowledge.

Semantic heterogeneity: It happens when sources define concepts in a different way. These conceptual differences may be granularity differences, if the same domain is described with different levels of detail, coverage differences, if the two domains are different but have parts in common, and differences in perspective, when the same domain is described with the same level of detail but from another perspective (e.g. political and topographic maps).

Terminological heterogeneity: It results from differences in the terminology used for the same entities. Synonyms, acronyms, different idioms, distinctive technical vocabulary, all contribute to terminological heterogeneity.

Semiotic heterogeneity: Semiotic heterogeneity is the most difficult to mitigate. It concerns to the way people around the world interpret a term in different contexts. A good example is the concept "ontology" itself.

2.3.1 Schema and Ontology Matching

One of the main challenges to solve is schema or ontology matching. Schema matching aims at finding the mapping between two different schemas. It differs from ontology matching by the amount of information available a priori. Schemas often do not provide explicit complex semantics for their data, so techniques to extract information encoded in a schema frequently complement the matching itself. Certain characteristics of the schemas involved have direct impact in the results:

Label complexity: The analysis of the labels from a schema/ontology plays a big role in a matching system. Although background knowledge sources help, very specific technical domains or the adoption of abbreviations heavily impact the performance of the system.

Structure complexity: Structure is also a good indicator of how hard the matching procedure will be for the computer, since almost all tools resort to its comparison to perform a match. Previously matching results may help with very expressive ontologies with various kinds of relationships and constraints between classes.

Definition 3. Given two schemas/ontologies, S and S', the matching is given by a set of alignments $A = \{a_1, a_2, ..., a_n\}$, $n \in \mathbb{N}$. Each a_i $(i = \{1, 2, ..., n\})$ is quintuple $a_i = (id, e, e', v, R)$. id corresponds to a unique identifier for a_i; e, e' are elements from both schemas/ontologies; v is a measure, typically in the range [0,1], that expresses a confidence level for that alignment; R is the relation holding between the two entities (e.g. equivalence $\equiv$, inclusion $\sqsubseteq$, overlapping $\sqcap$) [65][90][116].

Generally speaking, a schema/ontology matching algorithm receives as input two schemas/ ontologies and an optional input alignment set, a pre-mapping. Depending on the technique, other parameters and external sources (e.g. dictionaries) may also be introduced in the system. The output is the final alignment set, as Figure 2.11 depicts, and it is given by $A' = f(S, S', A, p, r)$, being S and S' the two schemas/ontologies, A an input alignment set, p and r the parameters and external sources, respectively [65].

Typically the matching process follows the following pipeline [12][65]:

Parsing: In the first step, inputs are read using an interpreter (e.g. SAX and OWL API), all the information is normalized and stored for future usage.

Lexical equivalence matching: The second step is dedicated to find equivalent labels.

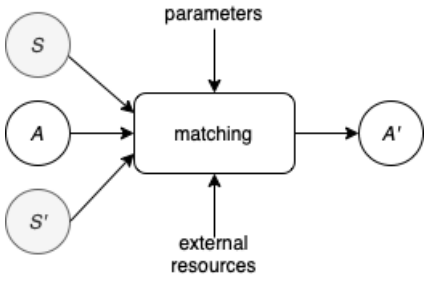

Figure 2.11: Matching Process (Adapted from [64])

Other matching techniques: Different strategies are put in place to map the entities not previously matched. This is where most tools differ, due to the variety of techniques available.

Combination and filtering: The outputs are combined if various techniques are used in parallel and the alignment set is filtered in order to produce a final verdict.

Reasoning and repair: This last stage only occurs if a tool to assess the coherence of the produced match is implemented.

It was mentioned above that structure actively impacts the matching. Having that in mind, different structures lead to a variety of different kinds of output alignments. The alignment cardinality of the alignment can be one-to-one ($1:1$), one-to-many ($1:m$), many-to-one ($n:1$) and many-to-many ($n:m$). If we consider the whole set of alignments (Figure 2.12), the multiplicity is denoted by $+$ if the match is total (all elements belong to an alignment), 1 if the match is injective and total (all elements belong to one and only one alignment), ? for an injective match (not all classes are matched but each alignment has cardinality 1:1), and $*$ if contrary (not all elements match and the local cardinality can be anyone) [65].

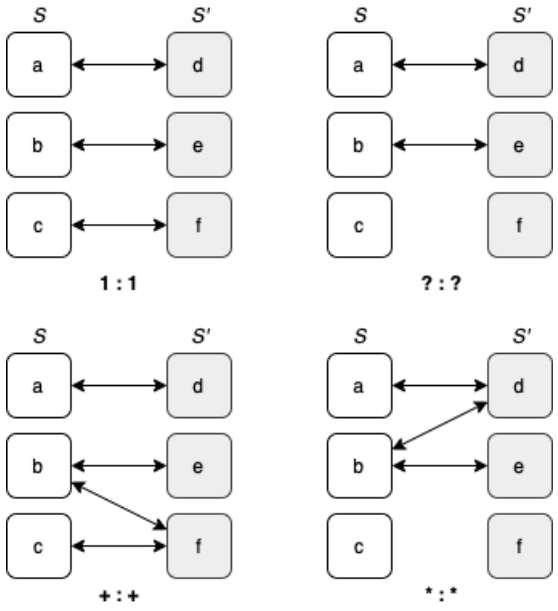

Figure 2.12: Examples of types of global cardinality (Adapted from [65])

2.4 Conclusion

In conclusion, this chapter introduced the basic concepts involved in this project. They are UBP's architecture and the origin of semi-structured data.

Although it did not totally succeed, the Semantic Web introduced some concepts that prevailed, for instance, schemas and ontologies, which became part of the web as it is today. Other visions of big data and machine learning technologies, for example, have emerged, which allowed the interpretation of that data. However, because there is not an agreed formatting standard between all sources, they remain heterogeneous and the integration dilemma persists. It is this problem that schema/ontology matching is concerned with. Such process is complex and it is not trivial nor direct, so heuristic methods are the choice for the majority of cases.

The next chapter makes an overview of several technique categories utilized in schema/ontology matching, as well as a few examples of known implemented systems, which served as inspiration for the development of the UBP's integration solution.

Chapter 3

State of the Art

The state of the art provides an understanding of the research that has been done so far about a certain topic. It helps the scientist being aware of what others have been working on, avoiding past mistakes and supporting future decisions.

This chapter provides information about the development of schema matching as a research field in the past years. The central subject matter thought is the categorization of the techniques commonly adopted by several systems, some of them presented later on. The chapter ends with a brief reflection on the knowledge acquired.

3.1 Taxonomy

As introduced in the last chapter, schema matching is a complex task often difficult to generalize. It highly depends on the domain, input (e.g. XML, relational model, ontology etc.) and how the output is meant to be shaped (e.g. final mapping, suggested matches displayed on a Graphical Interface (GUI) for an operator to validate etc.). Therefore the techniques used vary a lot and there is not a secret recipe nor guaranteed results.

Schema Matching saw its beginnings in the 1980s but, back then, it was addressed as part of a specific data integration problem [42][141][92]. During the 1990s, there was a boom of research done on the topic and, by the end of the decade, beginning of the 2000s, clear efforts were made to compile all the advances at that time [119][84][118]. These studies allowed the consolidation of generic schema matching and mapping as an independent research field. At the same time, the first generation of tools dedicated to perform schema integration were born, as observable in projects like Microsoft's Cupid (generic matching) [98] and IBM's Clio project[1] in collaboration with the University of Toronto (entity mapping) [106].

In their famous paper "A survey of approaches to automatic schema matching" [119], Rahm and Bernstein compiled and elaborated a classification for matching approaches, which has been the basis for the development of a solid taxonomy over the past years. For that, they considered several criteria worth to mention.

Individual or Combination: The two main categories distinguish the usage of one matching approach from the combination of multiple ones.

[1] https://researcher.watson.ibm.com/researcher/view_page.php?id=6965

Element or Structure: The data considered may be instance data or just schema information.

Language or Constraints: Usage of relationships or textual descriptions of schema elements.

Other criteria: Other criteria like auxiliary information and cardinality may also be accepted.

Such taxonomy tree can be illustrated by a scheme similar to Figure 3.1. The two main branches of the tree are specified into more depth in Subsections 3.1.1 and 3.1.2.

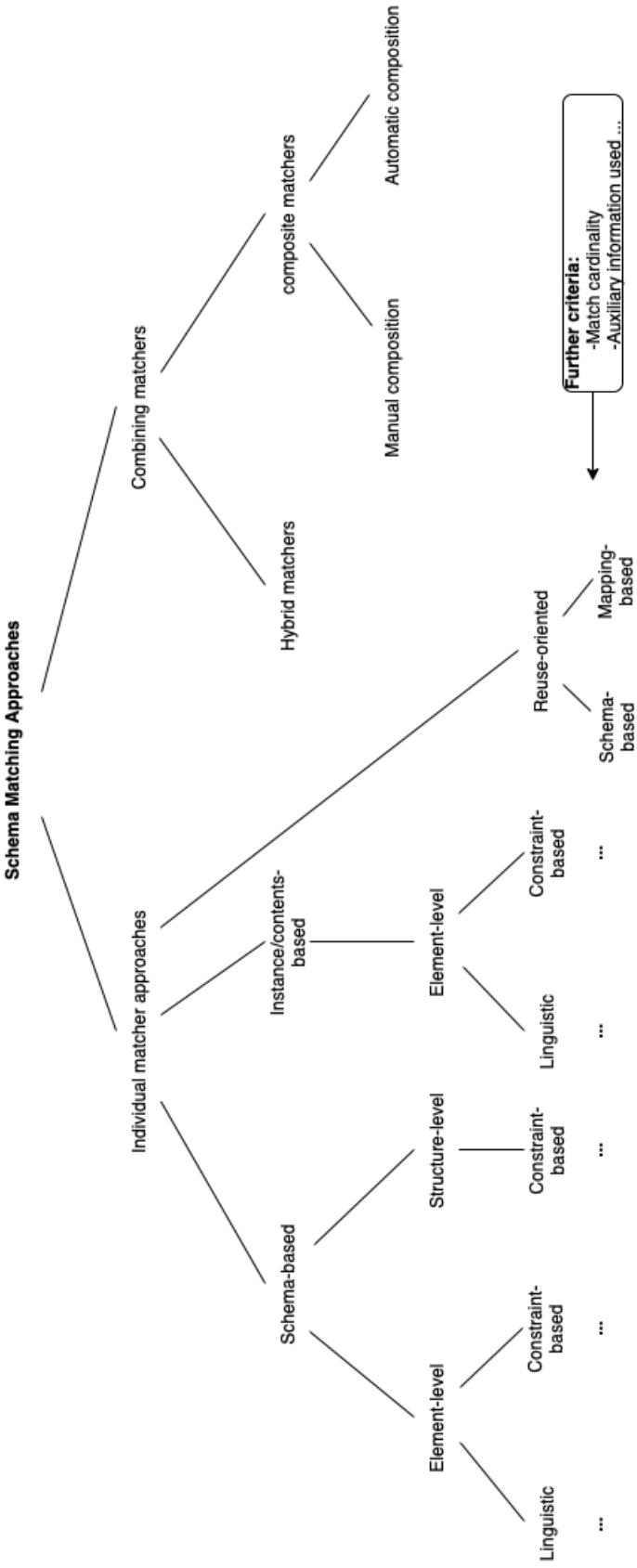

Figure 3.1: Taxonomy proposed by Rahm and Bernstein (Adapted from [119] and [76])

It is important to remark another classification diagram proposed by Shvaiko and Euzenat [65][137], where only individual matching approaches were taken into consideration. The diagram comprises three levels. At a core level, the basic techniques are differentiated, from string-based to graph-based. The outer opposite layers classify the basic techniques considering the kind of input and how the algorithm interprets that input (granularity).

3.1.1 Individual Matching

On the left sub-tree (Figure 3.1), individual approaches are divided according to specific characteristics they exploit when trying to acquire valuable information for the matching. Its child nodes help differentiating the different techniques by levels.

Schema-level: Schema-level matchers utilize the available information in the schema. This includes names, descriptions, restrictions and the structure itself [119]. In the taxonomic tree, this level can be broken down into two other levels. They depend on whether the matcher analyzes the schema structurally or element-wise.

> **Element-level:** It exploits the attributes of the schemas to be matched. At the finest level of granularity, it works with the atomic level attributes.
>
> **Structure-level:** Structure-level combines the context where each attribute is with the overall schema structure. In other words, it considers the surroundings of an element, for example, which is the father attribute or which other elements are present at the same level in the structure. This becomes really useful when uncertainties appear, due to the possibility of only a partial match.

Instance-level: On the other hand, instance-level matchers consider instance data when performing the match. They play an important role when schema information is not that vast [29].

Reuse-oriented: Rahm and Bernstein proposed a taxonomy tree that can be extended according to new criteria. Reuse-oriented techniques explore external resources, often collected in previous matching executions [76]. The two main sub-categories are as follow.

> **Schema-based:** Frequently used names can be stored in an internal global dictionary and used as a later reference. This principle works, not only for attributes, but also for entire schema fragments.
>
> **Mapping-bases:** Taking reuse even further, it is possible to benefit from previous mappings or similarity values in favor of a quicker and precise matching.

3.1.2 Combining Matching

The majority of systems are combined matchers, which means that several individual matchers are put together, complementing each other, in order to improve the results. The combining techniques are divided into the following classes, seen in Figure 3.1.

Hybrid Matching: Hybrid matchers combine several approaches in order to take the most of the characteristics of both schemas/ontologies. For instance, one can combine linguistic techniques with external sources [29].

Composite Matching: Different matches are performed independently and then the results are combined. Composite matchers are flexible enough to allow free ordering of

matchers. The matchers can be sequential, if the output of a matcher is the input of another, or simultaneous, if they execute at the same time [90].

3.2 Individual Techniques

The same procedures may be applied, for example, to build an instance-level matcher and an element-level one. It is up to the system architect to decide whether a technique is applicable to a certain situation. Nevertheless, all fall into one of the next categories.

3.2.1 Linguistic

Linguistic approaches are among the most common approaches. They are a helpful and indispensable resource for determining matches. As it is a vast category, sub-categories can also be identified.

Language-based

Language-based techniques are based on Natural Language Processing (NLP), making use of the morphological processes, such as inflection, the process of modification a base form in order to express number, gender, tense or person (e.g. $go \longrightarrow went$); derivation, which is creating new words from existing more simple ones (e.g. $treat \longrightarrow maltreat$); and compounding, when two independent lexical items are combined to form a new word (e.g. $god + parent \longrightarrow godparent$) [48]. These techniques are commonly applied before any other linguistic procedure, since it maximizes their results [65].

- *Tokenization:* Names are parsed into smaller yet meaningful tokens (e.g. $FirstName \longrightarrow First, Name$). The separators include punctuation, blank spaces, cases etc.

- *Elimination:* Unnecessary tokens are eliminated, such as prepositions, conjunctions and other stop words (e.g. $TheFirst \longrightarrow First$).

- *Stemming:* Stemming puts strings into their canonical form. It is done by removing or replacing sufixes. It is good at identifying common root forms among different words (e.g. $running \longrightarrow run$). The first stemmer was published in 1968 by Julie Beth Lovins [96], but perhaps the most popular is Porter's Stemmer [114] and its successor, Porter2. Lovins' algorithm removes the longest suffix from a word. The remaining is converted into a valid word via partial matching of words in a technical vocabulary. It can handle some irregular forms, but it frequently returns erroneous results. Porter is a rule-based stemmer, limited to English and It is widely used as it maintains a good balance between efficiency and simplicity. Since then, other contributions, such as Snowball[2] a framework for developing stemmers, has opened doors for new stemmers for different languages [115]. There are other less known stemmers available like Krovetz Stemmer (KSTEM) a hybrid technique (combination of dictionary and set of rules), ideal for using in conjunction with other stemmers, Paice/Husk's, an iterative stemmer, or Xerox's, that uses a lexicon database for mapping different word forms [140]. Despite all the advances, the perfect stemmer is still to be found.

[2] https://snowballstem.org

- *Lemmatization:* Just like stemming the goal of lemmatization is to reduce inflectional forms to root form, also known as lemma. Unlike stemming, lemmatizers are able to take into consideration the part of speech to perform a morphological analysis on a word, so is typically more powerful than stemming (e.g. *saw* $\longrightarrow$ *see*) [57]. Examples of tools that do lemmatization are Natural Language Toolkit (NLTK)'s WordNet lemmatizer[3], peharps the most well known, SpaCy's lemmatizer[4], TextBlob's[5], Stanford CoreNLP's[6], LemmInflect[7] and TreeTagger[8] [131][130].

String-Based

String-based approaches is another set of techniques. These are used to match entities from two different schemas. They rely on the basic principle that the more similar two strings are, the more likely they refer to the same concept [65].

- *Prefix:* Checks whether one of the strings to be matched is a prefix in the other string (e.g. *super* is a prefix in *supermarket*) [136].
- *Suffix:* Tests if an attribute's name is present at the end of the potential match string (e.g. $Name$ is present at the the end of attribute $FirstName$ so they likely refer to the same entity, or part of the same in this case) [136].
- *Equality of Names:* Another technique that goes in line with the previous ones is comparing the two strings to assess how similar they are. The similarity can be quantified using a simple measure like the Jaccard similarity coefficient, given by the formula

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{A \cap B}{A + B - A \cap B},$$

  where A and B are the two sets of characters and $J(A,B) \in [0,1]$, being $J(A,B) = 1$ if A and B are the same word. (e.g. *occupation* and *occupied* share the five first characters, therefore $J(occupation, occupied) = \frac{5}{8+10-5} \approx 0.385$)
- *N-gram:* N-gram [83, Chapter 3] is a sequence of N characters in a string. It can be used to compute the number of common character sequences in two strings. The more n-grams two words share, the higher the chances of them being semantically similar [109]. One could also apply the Jaccard formula presented above, or use instead the Dice Coefficient similarity [88]

$$DSC(A,B) = \frac{2 \cdot |A \cap B|}{|A| + |B|},$$

  where A and B are the two N-gram sets and $DSC(A,B) \in [0,1]$, meaning $DSC(A,B) = 1$ a comparison between the same word. (e.g. $S_3(occupation) = \{\mathbf{occ}, \mathbf{ccu}, \mathbf{cup}, upa, pat, ati, tio, ion\}$ and $S_3(occupied) = \{\mathbf{occ}, \mathbf{ccu}, \mathbf{cup}, upi, pie, ied\}$ share three 3-grams. Therefore, $DSC(S_3(occupation), S_3(occupied)) \approx 0.333$).

[3] https://www.nltk.org/_modules/nltk/stem/wordnet.html
[4] https://spacy.io/api/lemmatizer
[5] https://textblob.readthedocs.io/en/dev/quickstart.html#words-inflection-and-lemmatization
[6] https://stanfordnlp.github.io/CoreNLP/lemma.html
[7] https://github.com/bjascob/LemmInflect
[8] https://www.cis.uni-muenchen.de/ schmid/tools/TreeTagger/

- *Google Similarity Distance:* Comparing N-grams may not be a good approach when multiple domains share some common words. By using Normalized Google Distance (NGD) it is possible to calculate semantic distances based on the returned results in Google's search engine.

$$NGD(x,y) = \frac{max\{logf(x), logf(y)\} - logf(x,y)}{logN - min\{logf(x), logf(y)\}},$$

  where x and y are two terms to be compared, f is the function that returns the number of search hits, N is the number of web pages indexed by Google and log is a binary logarithm. It is also important to note that $NGD(x,y) \in [0, \infty[$, which means, the closer to 0, the more related x and y are (e.g. If, when searching for $horse$, the number of results was 46700000, the number of hits for $rider$ was 1220000, the number of pages where both terms occur was 2630000 and the total number of web pages indexed by Google was 8058044651, then $NGD(horse, rider) \approx 0.443$) [50].

- *Regular expressions:* Regular expressions, RegEx as abbreviation, describe a pattern in text. They are convenient in tasks like search, input validation, lexical analysis or word matching. In schema matching, regular expressions are useful when inspecting instances, mostly because attributes' instances, such as phone numbers or addresses follow a regular pattern (e.g. For the purpose of this example, lets simplify and assume all phone numbers begin with a thee capital letter calling code and are followed by a 5 minimum digit sequence. Then a valid regular expression for all phone numbers would be $[A-Z]\{3\}[0-9]\{5,\}$, in the POSIX[9] standard) [99].

- *Embedding:* Words and sentences can be represented in the form of vectors encoding their meaning. The closer they are, the more similar they are. Embeddings are trained on large texts such as books or internet pages, and one can choose to use a pre-trained model or build local embeddings for a specific domain [47]. Besides schema and entity matching, the variety of use cases where embeddings can be applied range from speech recognition [39] to image classification [71]. Popular algorithms for training embeddings include Word2Vec, GloVe, ELMo and BERT.

  Word2Vec is a family of architectures that can be divided in two main methods: Continuous Bag of Words (CBoW) model, which is specialized in prediction a target word based on other words around it; Continuous Skip-gram model, which predicts words before and after a given word [125][103][104].

  Global Vectors (GloVe), unlike Word2Vec, does not rely on local context words, so the semantic learnt is more broad, since it takes the whole set of texts into consideration, and not only the surroundings [111].

  Embedding from Language Models (ELMo) follows a bi-directional LSTM architecture and it is good at capturing a word's different levels of meaning, *i.e.* polysemy (e.g. "**park** the car" and "play at the **park**") [112].

  Bidirectional Encoder Representations from Transformers (BERT) is another alternative for extracting contextualized meaning of words. BERT is based on a transformers architecture [148], a newer alternative to LSTMs, and offers two options: BERT Base with twelve encoder layers and BERT Large, with twenty four layers, suited for more demanding tasks [55]. The input are sentences that are tokenized into different pieces within the delimiters $[CLS]$ and $[SEP]$. BERT is capable of processing unseen words, because it slices them into sub-words (e.g. "My embedded sentence." $\longrightarrow$ $['[CLS]', 'My', 'em', '\#\#bed', '\#\#ded', 'sentence', '.', '[SEP]']$ [10]).

[9] https://en.wikibooks.org/wiki/Regular_Expressions/POSIX_Basic_Regular_Expressions
[10] http://mccormickml.com/2019/05/14/BERT-word-embeddings-tutorial/

The output vectors can be compared using cosine similarity measure[11] to determine how distant two concepts are, based on the angle between the vectors. That is,

$$sim(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

where A and B are vectors, and A_i and B_i are their respective components.

In schema matching applications, over the last few years, new approaches have been exploring embeddings [32] mainly for extracting information from instance data [77][81].

- *Edit distance:* Edit distance is the number of operations needed to transform one word into another. There are several algorithms for calculating the distance such as Hamming distance, Longest Common Sub-string (LCS), Jaro and Jaro-Winkler variant, as well as Levenshtein and its variant Damerau-Levenshtein. Hamming distance only allows substitution, so it is not used to compare words of different lengths.

  LCS only allows insertions and deletions. It first finds the longest common sub-string and then calculates the similarity measure dividing the length of the common sequence, by the minimum, maximum or average lengths of the compared strings.

  Jaro and Jaro-Winkler are a family of commonly used metrics in entity matching related applications [150], which measure the common characters in two words. They are among the fastest algorithms [49]. In Jaro only transpositions are possible. The score is given by

$$sim_j = \begin{cases} 0 & , m = 0 \\ \frac{1}{3}(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}) & , m \neq 0 \end{cases},$$

  where s_1 and s_2 are the lengths of the compared strings, m is the number of matching characters and t the number of transpositions.

  Jaro-Winkler is based on Jaro distance but it favors strings with common prefixes.

$$sim_w = sim_j + lp(1 - sim_j),$$

  where sim_j is the Jaro distance between the two strings, l is the length of a common prefix (up to a maximum four) and p is a constant scaling factor, 0.1 by default.

  Levenshtein distance allows insertions, deletions and substitutions. There is also a variant of Levenshtein distance, called Damerau-Levenshtein, which counts transpositions too as single word edits [49].

  (e.g. Simple Levenshtein exercise[12]: The difference between *kitten* and *sitting* is **k***itten* $\longrightarrow$ **s***itten*; *sitt***e***n* $\longrightarrow$ *sitt***i***n*; *sittin* $\longrightarrow$ *sittin***g**, which makes a total of three edits. The output can be normalized by aplying such formula $1-(\frac{dist(s_1,s_2)}{max(|s_1|,|s_2|)})$, being the numerator the number of edits and the denominator the length of the longest string. Thus, $sim(s_1, s_2) = \frac{3}{7}$).

Phonetic

Phonetics are important in schema matching as they allow to bypass double spelling words and mistakes up to a certain point. Phonetic techniques encode a string according to the

[11] https://en.wikipedia.org/wiki/Cosine_similarity
[12] https://en.wikipedia.org/wiki/Levenshtein_distance

a, e, h, i, o, u, w, y	$\longrightarrow$	0
b, f, p, v	$\longrightarrow$	1
c, g, j, k, q, s, x, z	$\longrightarrow$	2
d, t	$\longrightarrow$	3
l	$\longrightarrow$	4
m, n	$\longrightarrow$	5
r	$\longrightarrow$	6

Table 3.1: Soundex table [49]

a, e, h, i, o, u, w, y	$\longrightarrow$	0
b, p	$\longrightarrow$	1
c, g, j, k, q	$\longrightarrow$	2
d, t	$\longrightarrow$	3
l	$\longrightarrow$	4
m, n	$\longrightarrow$	5
r	$\longrightarrow$	6
f, v	$\longrightarrow$	7
s, x, z	$\longrightarrow$	8

Table 3.2: Phonix table [49]

way it is pronounced. The main limitation of these techniques is their language dependence, because every language has different pronunciation aspects. Homonyms (words read and/or written in the same way but with different meanings, for instance *right* $\longrightarrow$ *correct* and *right* $\longrightarrow$ *direction*) can also be an obstacle to phonetic algorithms.

- *Soundex:* Designed for the English pronunciation, Soundex's history started in 1918 but surprisingly it has been a very popular algorithm. The idea is that the characters after the first letter are replaced by numbers (see Table 3.1). Adjacent letters with the same number are merged and the returned code is always four characters length, which means it may be shortened if the word has more letters or padded with zeros, if shorter (e.g. $judge \longrightarrow J320$).

- *Phonix:* Phonix [72] intends to be an improved version of Soundex, like many other that appeared after it. More than a hundred rules were added to make a pre-processing before the encoding using Table 3.2. Due to its complexity the algorithm is slow comparing to other approaches [49]

- *Double-Metaphone:* Double-Metaphone is the second version of another encoder, Mataphone, developed by Lawrence Phillips. It attempts to deal with non-English words [49]. Just like Phonix, has many pre-proccessing rules and the result are 2 codes only made of letters (e.g. $judge \longrightarrow [jj, aj]$).

- *Beider-Morse:* Beider-Morse Phonetic Matching (BMPM) is an algorithm that was designed to tackle the problem of false positives in Soundex based systems. It does so by taking into consideration, not only the sound of the words, but also their linguistic properties. It also supports ten different languages. BMPM has been proved especially useful, when comparing different spelling names [36][69] (e.g. names like Schwarz and Schwartz, two alternative German spellings[13]).

[13] https://avotaynuonline.com/2008/07/beider-morse-phonetic-matching-an-alternative-to-soundex-with-fewer-false-hits-by-alexander-beider-and-stephen-p-morse/

3.2.2 Reuse-Oriented

Most of the times, the information available for matching is insufficient. For this reason, the reuse of information previously obtained, whether it is a mapping or another external resource, is crucial not only to facilitate the matching process, but also to improve the matching quality.

Schema and Mapping-based

As introduced in the preceding section, reuse oriented techniques are mainly based on data previously analyzed, either it is parts of the schema or alignments already determined.

- *Alignment reuse:* When the two schemas/ontologies to be matched are similar to already matched schemas/ontologies or even too big, previous matches may be reused to expedite the process [38]. The simplest approach on alignment reuse is to compare the attributes to be matched with similar previous matches, also known as transitive matching (e.g. if $A \longrightarrow B$ and $B \longrightarrow C$, then it is likely that $A \longrightarrow C$). The transitive matching can be calculated by multiplying the two similarities or averaging them (e.g. if $sim(x, y) = 0.5$ and $sim(y, z) = 0.7$, then $sim(x, z) = 0.35$ if the multiplication method is applied [58])

- *Schema/Ontology reuse:* Related schemas/ontologies are also good pieces of information and their applications are varied. The use of a reference schema/ontology in the matching process is known to reduce complexity in multi-schema matching. The idea is that, by comparing all schemas individually with a reference ontology and then extracting the relationships between different schemas/ontologies from the first set of relationships with the reference schema/ontology [136][67].
  Previous matched and validated schemas may also constitute great training datasets, that can feed a machine learning algorithm. Some research has been done about this, such as Semantic Integrator (SemInt) [94], one of the first approaches using neural networks, Learning Source Descriptions (LSD) [59], detailed in the next section, clustering at an element or schema level [87][117], normally used for large scale schema matching [129], and Random Forest for Schema Matching (RF4SM), a family of techniques popular for its simplicity and input flexibility [124]. However, since most of these methods require a lot of training data, machine learning techniques are often too difficult to apply in the schema matching context.

- *Instance data reuse:* Instance data is one abundant resource, comparing to schema information. Hence, it is easier to employ instance data in the training of models that extract the characteristics of different attributes. The use of these models improves the ability to discover relationships between entities, since some attributes' instances may not have a clear structure defined by a simple regular expression. However, using instance data is not possible, for instance, when those instances contain users' individual sensitive [154] information and it is less effective when instances have high variability [82].

Other External Resources

The use of other external resources, for instance dictionaries, thesauri and ontologies, is a good way to complement the available information. They are also necessary when a deeper

semantic context is wanted [76]. Therefore, external resources go hand in hand with the use of other linguistic techniques.

- *Thesauri:* A thesaurus is a collection of words grouped by meaning [19]. Thesauri can be common knowledge, for general use, or domain specific thesauri, which may limit the usage but often offers better results (e.g. a thesaurus for geography terms) [136]. WordNet[14] is a good example of a lexical database where different concepts are put together as sets of synonyms (synsets) alongside a short definition and usage examples [65, Chapter 5][105]. From these sets it is possible to cluster words semantically [151] and extract other relationships between words, for instance antonomy, hyponymy and hypernymy, and holonymy and meronymy[15]. Such relationships allow, in the schema matching context, to conclude whether an attribute is a subclass of another [38, Chapter 3]. An alternative way of finding matching candidates is to compare two words' synonym sets [119], for example. Some similarity measures applicable to quantify semantic similarity are the Wu and Palmer's, Resnik's, as it will be detailed later when clarifying taxonomy-based techniques [107][132].

- *Upper level ontologies:* Upper level ontologies like Suggested Upper Merged Ontology (SUMO) and Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE) are a good external knowledge extensions [145]. These ontologies describe general concepts providing a basic semantic structure for ontologies specific for a domain.

- *Linked Data:* As the web grew, more and more datasets became available online. As explained in the previous chapter, structured data is connected forming a knowledge graph, a dense nest of linked data, such as DBpedia[16] and Wikidata[17], which can also be interlinked to compose even bigger clouds like the Linked Open Data (LOD)[18]. To manipulate data, RDF triples can be queried using SPARQL. Several attempts of integrating these resources as a way to enrich schema information have been made, such as [86][30].

3.2.3 Constraints

Constraints are a big part of a schema/ontology. Ranges, data types, uniqueness and relationships should provide good support in matching decisions. Despite adding valuable knowledge, constraints are not as effective when they are the only information level considered, because only considering data types, for instance, may lead to several misinterpretations of the schema/ontology.

Data Type

Data types are, many times, the only data available, besides attribute names and instances,

[14] https://wordnet.princeton.edu/

[15] Antonyms are words that denote opposite concepts (e.g. sad, happy). Hyponymy and hypernymy are relations that specify a concept generally or particularly, respectively (e.g. appliance is the hypernym of dishwasher and dishwasher is the hyponym of appliance). Holonymy and meronymy express a whole from a given part and a part form a given whole, respectively (e.g. body is the holonym of feet, legs and head, while teeth is a meronym of mouth)

[16] https://www.dbpedia.org/

[17] https://www.wikidata.org/wiki/Wikidata:Main$_{Page}$

[18] https://lod-cloud.net/#about

thus they represent a big part of constraint matching. Some data type comparison techniques can be seen below.

- *Compatibility Table:* The simplest way of comparing data types is to manually build a table, like Table 3.3, with all combinations and respective similarity values. However, this method is not objective nor reliable [62].
- *Data Type Hierarchy:* Another approach to determine the similarity between data types is to organize the data types in a tree structure. The similarities can be calculated as follows.

$$sim(x,y) = \begin{cases} e^{-\beta l} \times \frac{e^{\alpha h} - e^{-\alpha h}}{e^{\alpha h} + e^{-\alpha h}} & , x \neq y \\ 1 & , x = y \end{cases},$$

  where x and y are the two type nodes being compared, l is the shortest path length between x and y, h is the depth of the LCA, α and β are adjustable parameters, and $sim(x,y) \in [0,1]$. This measure is based in the distance between the two types in the tree and on the depth of the LCA, and supported by Shepard's law, a similarity function applied in many research fields [78]. More on this topic in [62]

Type x	**Type y**	**Compatibility (x, y)**
string	*string*	1.0
decimal	*float*	0.8
string	*date*	0.2
float	*integer*	0.9

Table 3.3: Data type compatibility table (Adapted from [98])

Graph-based

Graph-based techniques are a good example of the use of relationships between attributes within the same schema/ontology. Schemas/Ontologies are usually cast into a standardized format before matching. Usually they are converted into labeled-graphs, because graph theory has been extensively studied and the techniques for matching graphs are well developed. In a labeled-graph vertices represent entities and edges the relations between them [34]. The similarity between nodes is a result of the comparison of their positions in the two graphs.

- *Graph Matching:* Graph isomorphism is a classic problem studied for many years [51]. Due to its complexity (NP-complete), usually approximate methods are used to find an error-tolerant solution for graph or sub-graph matching. More on this topic here [152].
- *Children nodes:* One of the similarity criteria for matching nodes is the similarity between their non-leaf children.
- *Leaves:* The basic principle of this technique is that two nodes are assumed to be similar if their leaves are identical, even if the nodes in between are not.
- *Relations:* One can also look at the relations between nodes to infer if there is a high degree of similarity. If two nodes have all the same types of edge attached to them, the chance of them being similar too is high.

Taxonomy-based

Taxonomy-based techniques belong to structured-level techniques and the principle behind them is that subsets/supersets of similar sets are likely to be similar too [65].

- *Bounded path matching:* The match is done by cutting two paths of the schemas/ontologies and comparing the a term to another in the same position in the other path.
- *Super/sub-concepts rules:* The matcher takes into consideration the concepts above and beyond an attribute. If, for instance, two attributes' sub-concepts are the same, then they are similar too.

In spite of being primarily used for linguistic support, external sources such as WorNet can also be used to implement these techniques, as words, the nodes, can be connected with "is-a" relationships, the edges [132][134]. There are multiple similarity measures, but two basic categories of approaches stand out: edge-based, which use edge information and node-based, which use node information. An example of a very well known edge-based measure is Wu and Palmer's. Wu and Palmer similarity is based on the distance to the root and the distance to the common ancestor of the nodes that are being compared. Thereby, Wu and Palmer similarity measure is given by the following expression

$$sim(x, y) = \frac{depth(cca(x, y))}{depth(x) + depth(y)},$$

where x and y are the two nodes being compared, *cca* is a function that returns the closest common ancestor of two nodes, *depth* is a function that returns the distance from the root to a node, and $sim(x, y) \in]0, 1]$ (e.g. if *carnivore* is *cat* and *dog*'s closest common ancestor, $depth(cat)$, $animal \longrightarrow irrational \longrightarrow carnivore \longrightarrow feline \longrightarrow cat$, and $depth(dog)$, $animal \longrightarrow irrational \longrightarrow carnivore \longrightarrow canine \longrightarrow dog$, are both five, and $depth(carnivore)$ is three. Then $sim(dog, cat) = \frac{3}{10}$).
Perhaps the most used node-based measures are Resnik's and Lin's measures. Resnik similarity [123] considers the content of the compared nodes' closest common ancestor and it is given by

$$sim(x, y) = ic(cca(x, y)),$$

where x and y are the two nodes being compared, *cca* is a function that returns the closest common ancestor of two nodes, *ic* is a function returning the information content in a particular node, and $sim(x, y) \in [0, \infty[$.
Lin's measure [95] is a variation of Resnik's formula. The information content of the ancestor is enriched with the information content of the two compared nodes. It is defined as

$$sim(x, y) = \frac{2 \times ic(cca(x, y))}{ic(x) + ic(y)},$$

where x and y are the two nodes being compared, *cca* is a function that returns the closest common ancestor of two nodes, *ic* is a function returning the information content in a particular node, and $sim(x, y) \in [0, 1]$.

3.3 Combination Techniques

As mentioned before, several matchers can work together for performance improvements. The results of individual matchers require some sort of combination system, so that the similarity cube (the matrices with all similarity values between attributes) is reduced to only a bidimentional matrix and a final veredict for each pair is output, according to a threshold, as shown in Figure 3.2. The techniques presented bellow are the combining methods most commonly referred in the literature.

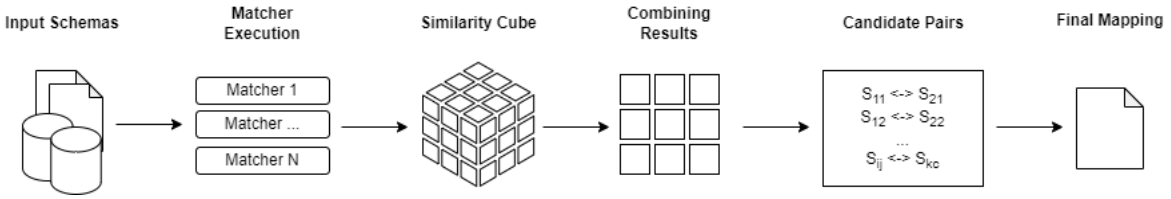

Figure 3.2: Combinatorial Matching Process (Adapted from [113])

- *Minimum:* This technique chooses, for each pair, the minimum similarity value computed by different matchers. For a value to be chosen, the others have to be much higher, that is why minimum is a pessimistic solution [76].

- *Maximum:* The maximum approach, on the other hand, chooses the highest value computed for a pair, which makes it a very optimistic method [76].

- *K-order:* For this aggregation operator, it is necessary to order all the values for a pair from the smallest to the biggest one and then the k element is chosen, usually the median [53].

- *Average:* The average is the simplest example of a weighted technique for the decision process, because considers equal weights for every output. It is one of the most used, due to the consistency of results, derived from the balance between the matcher's strengths and weaknesses [113].

- *AHP:* Analytic Hierarchy Process (AHP), developed by Thommas Saaty, is a method for analyzing complex decisions [126]. The idea behind it is the organization of the problem model in a structured way with a goal on top and several attributes beneath, each one with different levels of priority in respect to the goal, according to a scale from one (same importance) to nine (extreme importance). A matrix is then composed with all the relative priorities and a normalized vector by squaring the matrix and dividing the sum of all lines by the sum of each line. The final attribute ranking is obtained when multiplying the original matrix by the normalized vector. The weights can then be used in a weighted arithmetic mean, for instance, defined in the following equation.

$$sim_{agg}(x,y) = \frac{\sum_{k=1}^{n} w_k \cdot sim_k(x,y)}{\sum_{k=1}^{n} w_k},$$

where x and y constitute the pair for each the aggregated similarity is being calculated, n is the number of similarity matrices, w_k is the weight attributed to each matcher and $sim_{agg}(x,y) \in [0,1]$ (e.g. apply AHP given three matchers a, b, c that return three similarity matrices. Lets say the relative priorities are defined in a matrix

M. Notice that the elements below and above the main diagonal are reciprocal.

$$M = \begin{array}{c} \\ a \\ b \\ c \end{array} \begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 1 & 2 & \frac{1}{3} \\ \frac{1}{2} & 1 & \frac{1}{4} \\ 3 & 4 & 1 \end{bmatrix} \end{array}$$

The next step is to calculate the normalized matrix N.

$$\begin{bmatrix} 1 & 2 & \frac{1}{3} \\ \frac{1}{2} & 1 & \frac{1}{4} \\ 3 & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & \frac{1}{3} \\ \frac{1}{2} & 1 & \frac{1}{4} \\ 3 & 4 & 1 \end{bmatrix} \approx \begin{bmatrix} 3.000 & 5.333 & 1.166 \\ 1.750 & 3.000 & 1.750 \\ 8.000 & 14.000 & 3.000 \end{bmatrix}$$

$$Total = 3.000+5.333+1.166+1.750+3.000+1.750+8.000+14.000+3.000 \approx 13.083$$

$$N \approx \begin{bmatrix} \frac{3.000+5.333+1.166}{13.083} \\ \frac{1.750+3.000+1.750}{13.083} \\ \frac{8.000+14.000+3.000}{13.083} \end{bmatrix} \approx \begin{bmatrix} 4.316 \\ 6.308 \\ 1.640 \end{bmatrix}$$

Finally, the ranking is given by

$$R = M \cdot N \approx \begin{bmatrix} 1 & 2 & \frac{1}{3} \\ \frac{1}{2} & 1 & \frac{1}{4} \\ 3 & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 4.316 \\ 6.308 \\ 1.640 \end{bmatrix} \approx \begin{bmatrix} 17.478 \\ 8.875 \\ 39.818 \end{bmatrix}$$

The weights can be used for calculating the final similarity of a pair. If a, b and c return, for a certain pair (x, y), the similarity values 0.500, 0.580 and 0.690 respectively, the final similarity for that pair is calculated as

$$sim_{agg}(x, y) = \frac{17.478 \times 0.500 + 8.875 \times 0.580 + 39.818 \times 0.690}{17.478 + 8.875 + 39.818} \approx 0.625$$

Example adapted from [97]).

- *HADAPT:* This method is based on a measure called harmony, determined by the entries in matrices that are maximums both in their column and row. The match results with higher harmony value will also have a higher weight [113].

- *OWA:* Ordered Weighted Averaging (OWA) are a set of operators that allow the choosing of weights according to several established criteria. For that, all values for a pair are sorted and then the weights are assigned. Minimum and maximum combination methods are particular cases of OWA operators, for example, where the minimum or maximum is assigned with weight one and the remainder elements with weight zero [53][52].

- *Sigmoid:* Sigmoid combination acts as a filter. It processes the results adjusting them to a sigmoid function. This increases significantly higher similarity values and penalises lower ones [113]. The formula of a sigmoid

$$f(x) = \frac{1}{1 + e^{-t(x-s)}},$$

where t is the slope, s is the shifting factor and $f(x) \in [0, 1]$. These parameters can be ajusted for better results.

- *OpenII:* OpenII converts the similarity values from an $[0, 1]$ interval to an $[-1, 1]$ interval. Values bellow zero have a low confidence so they are penalised, whereas values above zero will weigh more [113]. For a given pair the aggregated similarity is calculated using the following expression.

$$sim_{agg}(x, y) = f^{-1}\left(\frac{\sum_{k=1}^{n} |f(sim_k(x, y))| \cdot f(sim_k(x, y))}{\sum_{k=1}^{n} |f(sim_k(x, y))|}\right),$$

  where x and y are the pair for each the aggregated similarity is being calculated, n is the number of similarity matrices, $sim_{agg}(x, y) \in [0, 1]$, f is the function to convert the interval $[0, 1]$ into $[-1, 1]$ and f^{-1}, its inverse, that transforms the interval $[-1, 1]$ back into $[0, 1]$.

- *Non-linear:* Non-linear combination first performs a weighted sum, showed by the first part of the expression. After that, a correlation between similarities is calculated and, depending on the result of the sum, the value is added or subtracted to the sum calculated in the first step. A constant λ is always present, so that the combined similarity does not escape the $[0, 1]$ interval [113]. For a pair x, y the combined similarity is given by the following formula.

$$sim_{agg}(x, y) = \lambda \sum_{k=1}^{n} w_k \cdot sim_k(x, y) \pm (1 - \lambda) \sum_{c=1}^{n} \sum_{k=1}^{n} sim_c(x, y) \cdot sim_k(x, y),$$

  where x and y constitute the pair for each the aggregated similarity is being calculated, n is the number of similarity matrices, w_k is the weight attributed to each matcher and λ is a constant, as explained earlier, and $sim_{agg}(x, y) \in [0, 1]$.

- *Machine Learning:* It is also worth to mention that machine learning can also be used to automatically determine the weights of individual matchers. Some examples of these techniques include Meta Learner from LSD which uses stacking for learning how to combine in the best way the individual predictions from different matchers [59], Schema Matcher Boosting (SMB) which uses boosting as the ensemble model [100], and Alignment Process Feature Estimation and Learning (APFEL) a bootstrapping-based approach that generates new samples from pre-validated ones [63].

After calculating the final similarity matrix is necessary to return a final verdict for each pair. For that, it is important to determine a threshold that separates the matches from the non-matches. Some of the classic approaches include choosing the top N matches (ideal for $n : 1$ cardinality), MaxDelta, which returns the top candidate plus all others differing in similarities up to a certain value [58], and manual threshold, such as Cupid [98], that requires a specialist to input the desire threshold. Other approaches include supervised learning [32], active learning, the use of reward functions and statistical analysis, such as using bivariate normal distributions (representing the frequencies of hits and misses) to find the best threshold, so that false positives and false negatives are minimized [142].

3.4 Evaluation Metrics

Matching schemas/ontologies is impossible to perform always without an associated error, in the light of the present technology. The complexity of the process also makes the comparison between purposed solutions much harder. Therefore, it is important to quantify, in a standardized way, how close the obtained results are to the expected ones. The following

metrics are often used in the area to quantify how close the results are to the ground truth. They are based on the notions of true/false positives and true/false negatives, as highlighted by the formulas [37].

- *Precision:* The precision calculates the proportion of discovered matches that are truly relevant.

$$Precision = \frac{tp}{tp + fp},$$

where tp is the number of true positives, fp the number of false positives and $Precision \in [0, 1]$ (e.g. if $tp = 30$ and $fp = 20$, then $Precision = \frac{3}{5}$, which means 60% of the discovered matches are relevant).

- *Recall:* Recall, also known as sensitivity, is the proportion of relevant matches that were returned with respect to all relevant matches.

$$Recall = \frac{tp}{tp + fn},$$

where tp is the number of true positives, fn the number of false negatives and $Recall \in [0, 1]$ (e.g. if $tp = 30$ and $fn = 10$, then $Recall = \frac{3}{4}$, which means 75% of the relevant matches were returned).

- *F-measure:* F-measure, or f-score, is the harmonic mean of precision and recall. It has the disadvantage, however, of not showing when these two metrics are imbalanced.

$$F{-}measure(\alpha) = \frac{(\alpha^2 + 1) \times Precision \times Recall}{(\alpha^2 \times Precision) + Recall},$$

where α is a parameter often set to one and $F{-}measure \in [0, 1]$ (e.g. if $\alpha = 1$, $Precision = 0.6$ and $Recall = 0.75$, then $F{-}measure(1) = \frac{(1^2+1)\times 0.6\times 0.75}{(1^2\times 0.6)+0.75} = \frac{2}{3} \approx 0.667$. In this case, there is a good balance between $Precision$ and $Recall$, not one being much higher than the other).

- *Runtime:* Runtime quantifies the time a program takes to run. Besides runtime, knowing execution time of each operation is useful when trying to optimize the code. Units may vary depending on the type of program and resolution of the machine measuring it (milliseconds, minutes, hour etc.).

3.5 Systems

Over the years, data integration became a daily-life challenge for the ones working with large volumes of data. This includes industries like E-commerce, large companies with lots of services and even researchers from biomedical sciences. Consequently, different researchers and institutions sought for their own solutions. After the first attempts, more and more tools and comparative studies were published, and schema/ontology matching and mapping started growing roots as the problem was formalized [102]. Later, the introduction of other techniques that explore linked data and machine learning models allowed taking advantage of large volumes of data [30][41]. In the recent years, as formats like JSON gained popularity, new other tools also came to fill the gap [107].

Although a lot is still to be done regarding the evaluation and comparison of techniques, some advancements such as the Ontology Alignment Evaluation Initiative (OAEI) have

been a success. OAEI[19] is an international initiative that was founded with the goal of creating a singular and consensual evaluation of ontology matching systems. Not only does this aids in choosing the best approaches, but also their developers, who actually get a report of identified weaknesses of their systems. Every year, a conference is organized and a paper is published with a sum up of all evaluated tools.

Previously in this chapter, techniques taking advantage of several schema/ontology features were clarified. The following paragraphs are dedicated to cover relevant historical and state of the art tools that apply some of the above principles and methodologies. Table 3.4 synthesizes each system's features and highlights the main differences between them.

3.5.1 Cupid

Cupid is a hybrid system created by Jayant Madhavanm, Philip A. Bernstein, and Erhard Rahm. Despite of being one of the first attempts to solve generic schema matching, it is still a reference for the newer systems. It consists of three main phases. First, XML and relational schemas are imported and the attributes are normalized by performing tokenization, elimination of stop words, extra spaces etc. Abbreviations and acronyms are expanded too. An auxiliary thesaurus is key for this step. After that, the system categorizes all the elements in each schema, clustering them into categories based on characteristics such as data types and hierarchy inside the schema. A comparison is then done between elements of the two schemas from similar categories, extracting synonym and hypernym relationships, helped by a thesaurus. These three steps constitute the linguistic phase. The second phase corresponds to the structural matching. First inputs are converted into trees. The structure matcher is based on the intuition that leaf nodes encode much of their ancestors' information. The algorithm calculates structure similarity for leaf nodes by evaluating their data types compatibility. For non-leaf elements the structural similarity between elements of the two schemas is determined as the fraction of leaves that have at least one strong link to another leaf in the other element sub-tree. Two leaves share a strong link if the weighted sum between their linguistic and structural similarities (weighted similarity) stays above a given threshold. The algorithm also rewards or penalizes whenever two elements' weighted similarity is above or below a given threshold. The third phase uses the weighted mean between linguistic and structural similarities of each pair to produce the output, a set of $1:1$ leaf mappings, according to a threshold provided by the user [98][119][76].

3.5.2 COMA

Combination of Matching Algorithms (COMA) is a composite matcher, originally developed by Hong-Hai Do and Erhard Rahm. Since its release in 2002 the system has suffered two main updates, named COMA++ [31] and COMA3.0 [101][3]. In the latest major iteration, COMA3.0, there was even launched a business version, besides the open source version already available. Due to space reasons, only common features to all the versions will be discussed in detail. The system's core is divided in three modules: storage, matching and mapping. In newer versions a new interface module was also introduced, with support for an API and a GUI. The storage module, imports the XML schemas and relational schemas, ontologies (new versions) and the external resources (thesaurus and previous mappings) mapping schemas/ontologies into DAGs. The matching module consists of a library of elementary, hybrid and reuse-oriented matchers that can be configured by a configuration engine. The simple matchers include a matcher to look for common

[19] http://oaei.ontologymatching.org

prefixes and suffixes, a N-Gram matcher, an edit distance-based matcher, which uses the Levenshtein measure, a phonetic matcher implementing Soundex, a synonym matcher that uses a thesaurus to search for related words, and a data type matcher that follows a reference table specifying the compatibility between the different types. User can also provide input feedback in order to refine the matches. Hybrid matchers are combinations of different simple matchers. A Name matcher which resorts to affixation, N-gram, synonyms and Levenshtein distance to perform the matching. A TypeName matcher that computes the similarity based on the Name matcher and data type matching. NamePath is a matcher that combines name similarity (Name matcher) with schemas' strutural characteristics. This is achieved by joining all attribute names from each path in a single string and then using the Name matcher to compare these strings. There are two more hybrid matchers based on the structure of the schemas/ontologies, which are Leaves and Children matcher. The Children matcher considers, of two non-leaf nodes, the similarity of their children, which is recursively calculated from the similarity between their respective children. The similarity between the leaf elements is computed by the TypeName, as a default. On the other hand, the Leaves matcher only considers the leaves of the two compared nodes. This strategy helps reducing the computational cost, but it is also more stable in cases of structural conflicts. The last matcher is the reuse-oriented matcher, which showed to be important for improving matching results. Previous matches are stored in a repository and they are used to calculate transitive similarities. The similarities for all pairs are calculated and then combined. Finally, the mapping module carries out the final processing before exporting the mappings. This is also where ontology merging and manual mapping correction takes place in newer versions. Among the similarity aggregation strategies are the minimum, maximum and average, although the average has shown higher consistency as it better compensates defects of different matchers. The returned results consider mainly a $1:1$ cardinality [76][90][136]. Overall, the authors claim that the best configurations are the ones with multiple matchers, exploring different parts of the sources (structure, elements, constraints etc.) [58].

3.5.3 LSD

LSD is a composite system, developed by AnHai Doan, Pedro Domingos and Alon Levy, for matching XML sources. The system relies on several machine learning matchers that explore distinct parts of the schema, like the attribute names and word frequencies. Just like all supervised techniques, there is a learning phase and after that a classification phase. The learning phase comprises five learners, being them nearest neighbor Whirl learner, a Naive Bayesian learner, name matcher, county-name recognizer and Meta Learner. Nearest neighbor Whirl learner classifies an instance based on the labels of its neighbors in the training set. According to the authors, this matcher performs best when the elements have long textual values as instances. The Naive Bayesian matcher learns from word frequencies and it suits cases where there are words that clearly distinguish an entity. On the other hand, it does not work well for short numeric values. Name matcher is responsible for learning to detect synonyms. However, it does not work well for ambiguous names (e.g. *office* and *office_phone*). Another matcher is the County-Name Recognizer, which matches a certain entity (in this case a county) using external domain specific sources. These models are applied to data during the classification phase. The last learner is the Meta learner, introduced previously in this chapter, designed for automatically determining the weights when combining results. The application phase applies the trained models for computing the correspondences. Because there are several matchers involved in the process, there is also a final phase where the results are combined using by the Meta matcher. The matches produced have $1:1$ cardinality [59][119][76]. LSD modularity allowed other systems to be

developed on top of the original system. Such extension examples include Glue [61] and iMAP [56].

3.5.4 JSONGlue

JSONGlue is a hybrid tool designed by Vitor Marini Blaselbauer and João Marcelo Borovina Josko. This system stood out for being one of the only approaches dedicated specifically to match JSON sources. The system is made of four modules. The first module is the normalization module which is divided into three steps. In the first step the JSON files are imported and any irrelevant characters from the attribute names (extra spaces, stop words, punctuation etc.) are removed. The second step is the transformation of attribute names if there is a naming standard convention file available. The third step is responsible for building a disconnected graph composed of the imported schemas' sub-graphs, to facilitate the rest of the process. The second module is the linguistic one, which is responsible for measuring the string similarity between attributes of the schemas that are being compared. This is done using the Jaro-Winkler similarity described earlier in this chapter. For each comparison, the algorithm produces an edge with the similarity value, linking the two compared nodes in the graph. The third module, the semantic module. It uses WordNet[20] to look for the synset with the highest similarity for each pair of nodes. The similarity is calculated using Wu and Palmer explained in further detail too. Just like in the linguistic module a new edge is created for each pair in the graph. Finally the last module is the instance-based, which is divided into three steps. In the first step, the difference between attributes' average instance length is calculated. The second one also calculates a difference but it considers the standard deviation. The last step is responsible for building histograms for the characters' frequency and determining the distance between them. The system's current iteration does not include automatic threshold determination nor calculates the combined similarity results. The output is therefore the bipartite graph of $n : m$ similarities [43].

3.5.5 LEAPME

Developed by Daniel Ayala, Inma Hernandez, David Ruiz and Erhard Rahm, LEArning-based Property Matching with Embeddings (LEAPME) is a hybrid state of the art system that uses embeddings to help in the matching process. It supports many sources in an undefined number of formats, despite the first instance feature extractor working with JSON. The first step is the extraction of instance features. three hundred and twenty nine features are computed in this phase, three hundred of which are from the embeddings of the instances and the other twenty nine for other features such as punctuation, separators, numerical characters, token frequency etc. with the help of the semantic labelling system Tapon [33]. Then, the attribute features of the are calculated by averaging the previous instance features and complementing it with the average embeddings vector of the attributes' names. After that, features for each attribute pair are obtained by calculating the difference between the vectors of the two attributes that form the pair and another eight features dedicated to traditional string similarity. The final six hundred and thirty seven features comprise all of the previous features. After the feature extraction, the vectors are fed into a machine learning based classifier that will attribute weights and define thresholds automatically. Finally, a knowledge graph is generated based on the classifier results. Since many sources are supported, the final matches can be $n : m$. This approach showed to be

[20] https://wordnet.princeton.edu/

effective even with pre-trained word embeddings from other domains, which opens doors for the future use of transfer learning whenever training data is not available [32].

3.5.6 It's AI Match

"It's AI Match", developed by Benjamin Hattaschis, Michael Truong-Ngoc, Andreas Schmidt and Carsten Binnig, is a composite state of the art tool that aims at finding matches with embeddings. The inputs are relational tables and it consists of two main steps. The first phase is dedicated to shorten the search space when finding possible candidate tables. Table matching offers three different configurations. A structure based matcher that encodes table names, and attribute names into vectors. An instance based matcher that utilizes the instances. Encoding all instances may be very expensive, so a possibility is to choose only a few instances. Sampling may be done by selecting only the distinct instances, choosing N random samples, or choosing the most common instances. Besides instance and structural matching, it is also available a third configuration, which combines the previous two. After the first phase, there is an optional candidate verification, in which a human operator can add or remove matches. The second step is attribute matching. The goal is to find the final matches by comparing candidate attributes, produced in the first step. There are three matchers available. A name based matcher, that uses embeddings to represent and compare attribute names. An instance based matcher that works with the instances, similarly to the table's instance matcher. A third matcher, dedicated to extract information from comments that might exist in the schema. To the returned matrix a threshold is applied or several of these matchers' results can be combined. The output is a set of match results, $1:1$ or $1:m$. The authors also suggest that this tool is used as a supplement of other techniques [81].

3.5.7 Smat

Smat, developed by Jing Zhang, Bonggun Shin, Jinho D. Choi and Joyce C Ho, is an individual schema-level state of the art approach, that uses deep learning for schema matching. It consists of four modules. First, the inputs, collumn names and descriptions in relational tables, are tokenized by a Byte-Pair Encoding (BPE) and embedded into vectors. After that, Bidirectional Long Short Term Memory (BiLSTM) networks are used to capture the semantics in the description and names. BiLSTMs have the advantage, compared to LSTMs, of being able to use previous and future information in a sentence to better extract its semantics. The third step is the Attention-over-Attention (AOA) mechanism, which models the correlation between each attribute name and its description. For that, a pair-wise interaction matrix between attribute name words and description words is first calculated. Then, a column-wise softmax is applied to the interaction matrix to obtain the "column name to description" attention and a row-wise softmax followed by a column-wise averaging is applied so that column name attention is obtained. The final attribute attention matrix is obtained by computing the scalar product between "column name to description" attention and the transpose of the inverse column name attention attention matrix, i.e. a weighted sum of each individual "column name to description" attention. To predict whether two columns represent the same concept, the attributes' final attention matrices are concatenated with the difference between the two max-pooled attribute description matrices and input into the fourth and final step of the system, which consists of several fully-connected layers and a softmax layer. The output returned has $n:1$ cardinality. It is important to note that data augmentation and Controlled Batch Sample Ratio (CBSR) were also used to better deal with class imbalance. The fact that Smat does not

work with instances, allows its applicability to scenarios where sensitive data is involved, such as banking or healthcare [154].

	Approach	Input	External Sources	Techniques	Cardinality	GUI
Cupid	hybrid	XML, relational, thresholds	thesaurus	structural, constraints, linguistic, reuse	$1:1$	No
COMA family	composite	XML, relational, OWL, user hints, match combination	thesaurus, previous mappings (optional)	structural, constraints, linguistic, reuse	$1:1$	Yes
LSD	composite	XML, match combination	training data	constraints, linguistic, instances, reuse	$1:1$	No
JSONGlue	hybrid	JSON, thresholds	thesaurus, naming standards (optional)	linguistic, instances, reuse	$n:m$ (despite not processing the similarities)	No
LEAPME	hybrid	set of property instances, JSON (not confirmed)	training data (optional)	linguistic, instances	$n:m$	No
It's AI Match	composite	relational, thresholds, user hints, match combination	training data	structural, instances, schema information, reuse	$1:1$, $1:m$	No
Smat	individual	relational (column names and descriptions, reuse)	training data	linguistic, schema information	$n:1$	No

Table 3.4: Summary of the systems' characteristics

3.6 Conclusion

Schema/ontology matching first appeared associated with data integration and specific use cases. Although it still often appears associated with other integration tasks, schema/ontology matching gained importance, due to an increment in heterogeneous sources that need to be integrated, and now occupies more space as an independent and context decoupled research field.

This chapter introduced and described some of the techniques used in the schema/ontology matching process, following the core of Rahm and Bernstein's suggested taxonomy [119]. The techniques which support attribute correspondence, are imported, most of the times, from other scientific areas, including NLP and graph theory.

Matchers can be based on one single technique or may combine different techniques. Most systems implement combined matchers, because it allows them to explore different characteristics of the schema/ontology and, therefore, producing better results.

Although new advanced machine learning approaches to extract similarities, set automatic thresholds and weights have emerged, classic techniques are still very popular. Moreover, the seek for new generic systems seem to have slowed its pace over the last decade. The bibliographic review presented in this chapter also demonstrates lack of systems working with JSON format sources.

With all this in mind the next chapters propose and discuss the planning, requirements, architecture, implementation and testing of a solution for integrating data arriving Ubiwhere's UBP.

Chapter 4

Planning and Methodology

The good outlining of a project is key to its success. Project goals serve as reference during implementation, avoiding wastes of time, budget overruns and the identification of possible obstacles contributes to risk mitigation early on.

This chapter is focused on describing the methodology for this dissertation, as well as presenting the planning for all phases of execution. Firstly, the methodology chosen for this project is described. Then, the planning is presented, split into two different parts, corresponding to the two semesters of the year. The definition of success criteria and management of risks identified in preliminary studies end this chapter.

4.1 Process Management

During the course of this project, an Agile-based methodology was adopted. Agile is an iterative software engineering technique, in which requirements are divided into modules for an adaptive development [133]. Contrary to Waterfall methodology, Agile processes require less planning, avoiding micromanagement and providing more flexibility to the project. These characteristics benefit collaboration with the advisors, since there were scheduled weekly meetings with the project manager and a monthly meeting with both advisors to sum up everything done in that particular month. In these meetings the work done was discussed, and the next steps shaped around that. This keeps the project under control without ever curbing the intern's autonomous work.

Product Owner: Ubiwhere (represented by Ricardo Vitorino)

Project Manager: João Garcia

Developer: Jaime Marques

For the organization of this project GitLab[1] was chosen as the main tool. Not only does it offer a wide range of tools, from version control to a Kanban table for better visualization of the tasks, but it was also used for previous implementations of the UBP, which is an advantage when adding features to the platform. Furthermore, the time spent every week, was recorded on Easy Redmine[2], a solution for project management used by Ubiwhere to evaluate project costs.

[1] https://gitlab.com
[2] https://ubiwhere.easyredmine.com

4.2 Planning

The internship comprises two main phases organized in two semesters. This section presents the planning for both semesters, what aspects are approached first and at what time. There is a lot of debate around whether Gantt diagrams should be used or not in Agile-inspired methodologies [144][143], since they are very rigid and need constant updates. Nevertheless, it is safe to say that Gantt diagrams are used by many companies as they can be a great addition to the project from a software engineering standpoint. So, in this intermediate planning document a Gantt will be used to illustrate the past semester and the tasks for the second one will be displayed in a Kanban table.

4.2.1 First Semester

The first semester started in September 2021 and ended in February 2022. The tasks are summarized in Figure 4.1.

Intermediate Report: This set of tasks corresponds to all sub tasks related to the intermediate report. Firstly a draft of the report was made so that a structure was previously defined and agreed on. Eventually, the reported was written and delivered on 24th January 2021. After that, the presentation to the jury was prepared.

Project Management: The planning was part of a first iteration of work. Risk Assessment and management were present as the problem became well known and the obstacles for the project became clearer.

Research: Research represented the majority of the work developed in the first semester. The problem to be solved was initially identified and the literature review was done around that.

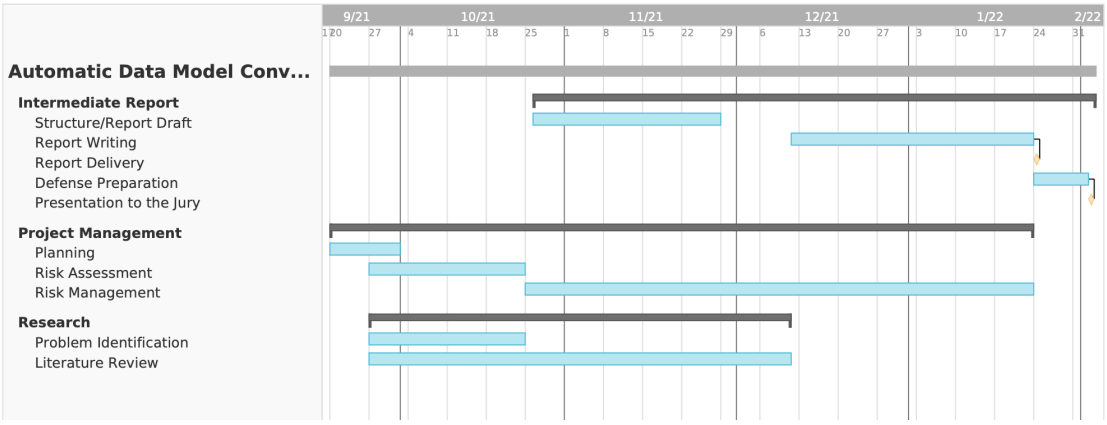

Figure 4.1: Gantt diagram of the first semester

4.2.2 Second Semester

For the second semester, the workload corresponds to the specification of the requirements, development and testing of the solution, as well as writing the final report. The tasks are condensed in a Kanban-style table on the GitLab project created earlier. Kanban tables

are a great way to display and group different tasks, make changes to them as the project progresses, but also keeping estimates and deadlines. The following table (Table 4.1) reproduces the Gitlab board for a better visualization of the tasks. As the project advances and different challenges arise, new sub tasks will probably be added. This flexibility is one of the biggest advantages of using a Kanban table and it was the main reason why it was chosen for the second semester over other planning methods like Gantt charts.

Task			**To Do**	**Doing**	**Done**
Category	**Description**	**Deadline**			
Report Review	Planning document	Mar 1			X
Report Review	Organize sections	Jul 15	X		
Report Review	Improve SoA conclusion	Jul 15	X		
Report Review	Explain how schema matching helps mitigate some of Semantic Web's issues	Jul 15	X		
Report Review	Math correction	Jul 15	X		
Report Review	Add new techniques	Jul 15	X		
Preparation	Study UBP models and importers	Feb 20			X
Preparation	Find test datasets	Feb 25			X
Preparation	Decide how to test the solution. thresholds and the metrics to be used.	Feb 25			X
Preparation	Analyze the datasets and decide the techniques that will be implemented	Mar 4			X
Documentation	Write the requirements	Mar 10		X	
Documentation	Draw the architecture diagram	Mar 15	X		
Development	Importing and pre processing	Mar 30		X	
Development	Calculation of similarity values (matchers)	Apr 30	X		
Development	Combination of results	May 15	X		
Development	Final mapping	Jun 15	X		
Development	Test the solution	Jul 1	X		
Final Report	Write the final document	Aug 1	X		
Final Report	Review of the final document	Sep 1	X		
Final Report	Preparation for the final presentation	Sep 5	X		

Table 4.1: Reproduction of the task table (March 2022)

The tasks were grouped in five main categories to help organize them.

Report Review: Tasks related to the report review include initial corrections based on the jury's feedback. This planning document is also included.

Preparation: Preparation tasks including final studies of the platform and techniques in order to decide which solution is going to be developed.

Documentation: Elaboration of support documentation for development.

Development: Development is the most important group of activities. This group of tasks aims at implement the requirements written earlier. Testing is also present, since the solution needs somehow to be validated.

Final Report: Final Report is the group responsible for all activities related to the composition of the final document and presentation to be delivery to the jury.

4.3 Success Criteria

The Success criteria is the agreement with Ubiwhere to ensure all guidelines are respected. To assess the success of this project, the following criteria were established:

- The specified requirements and architecture are approved by Ubiwhere.
- A minimal set of functionalities, also approved by Ubiwhere, marked as "Must Have" has to be implemented and tested.
- The final product must fulfill the functional requirements and quality attributes defined.

4.4 Risk Assessment and Management

There is a line separating a successful project from a failure. That line is embodied by a minimum set of conditions that must be met so that the project is considered a success, the Threshold of Success (ToS) [8]. ToS pictures, not only what is a successful project, but also what does not represent the success of a project, its failure, as a strategy to achieve good results. Those potential failure scenarios are risks. There are a lot of definitions of risk, but maybe the most common one defines risk as the probability times impact [157]. This rather simplistic definition has the advantage of attributing a number to risks, allowing us to sort them. The risks can then be displayed on a risk matrix, where the x-axis is the impact and y-axis the probability. Depending on the level of granularity one wants to achieve, these scales should be divided into different zones. In this case, both impact and probability are only separated into three levels, because it is simpler and there are not that many risks to justify sharper differences between them. For impact the levels are *Low Impact* (marked as 1), *Medium Impact* (marked as 2) and *High Impact* (marked as 3). For probability we have *Low Probability* (marked as 1), if the probability is less than 40%, *Medium* (marked as 2) when between 40% and 70%, and *High* (marked as 3) if it is above 70%. Risk assessment for this project is showed in Table 4.2 and Figure 4.2.

Another table was created for the mitigation strategies (Table 4.3). Mitigation plans help reduce the impact of a risk before it is too late.

When planning a project, having a global view of what will be done, how and when is important and provides valuable hints for what parts will require special attention. The planning divided this project in two semesters. Identification of the risks and possible mitigation strategies were made after that and made clear what challenges will possibly appear and what can be done to overcome them and maximize the chances of project success.

ID	Risk	Probability	Impact
R1	Lack of experience using technologies involved in this project.	3	2
R2	A task exceeds the time expected for its completion.	2	2
R3	Forced modifications in requirements and architecture.	2	3
R4	Schedule and workload conflicts with other university courses.	2	2
R5	COVID-19 pandemic restrictions.	2	1
R6	Not enough datasets to train/validate/test the solution.	3	1
R7	Too complex cases.	3	2
R8	One matcher does not work as expected.	2	2
R9	Not enough computational resources available.	2	2

Table 4.2: Risk assessment

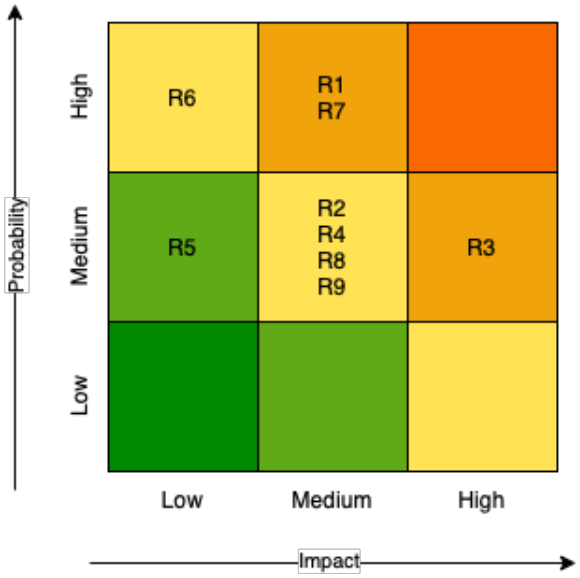

Figure 4.2: Risk Matrix

ID	Mitigation Plan
R1	More time should be allocated for learning the technologies necessary for the project.
R2	The delays have to be reported to the advisors, so that the task is reassessed and, if necessary, new priorities are established.
R3	Reevaluation of the requirements to keep the attention on those that really have high priority.
R4	Explain the situation to professors and negotiate alternatives.
R5	Alternative communication model agreed between the intern and the advisors.
R6	Avoid techniques that require a lot of training data such as machine learning techniques. Create new test cases manually.
R7	Limit the matching to a specific domain such as parking occupancy. Pre process the data in favour of an easier matching process.
R8	Consider at least one backup matching technique for every category implemented.
R9	Avoid computationally expensive techniques, opting for traditional ones.

Table 4.3: Risk management

This page is intentionally left blank.

Chapter 5

Requirements Specification and Architectural Decisions

Before the implementation it is important to gather and specify the requirements from the client. Requirements help the team to fully comprehend the problem, so that priorities are set in a pragmatic way, risks mitigated, and misconceptions avoided. It is also important to mention that requirements help managing expectations, as they represent a negotiation with the stakeholders. Therefore, to ensure high quality software, it is imperative that the specification of these requirements is unambiguous. Architecture definition is what comes after and it looks for the best system configuration that meets all the requirements.

This chapter specifies the requirements gathered from stakeholders, then presents the architecture of the solution, and later on it enumerates the technologies selected to implement the solution.

5.1 Scope and Stakeholders

As mentioned in the first chapter, this work is about integrating new data sources into UBP's data models automatically, without interfering too much with the system's pipeline. Hence, it was necessary to create a component that would fit the space occupied by manual conversion (after data import and before the entity mapping). To achieve such goals, it was decided to approach the subject as a schema matching problem.

Regarding the stakeholders involved, Ubiwhere was the only collaborator on this project. The company's role was to provide important feedback, always indicating the direction wanted for this project. The final result should be a relevant piece of software that meets the requirements previously established.

5.2 Functional Requirements

Functional requirements represent the necessities a software needs to fulfil through its operations and services. In addition to their definition, it is necessary to characterize them according to their priority degrees. For this, techniques such as the *MoSCoW* method can be used [10]. *MoSCoW* consists of assigning one of four levels to each attribute. The different levels are presented below.

Must Have (M): The requirements marked as *Must Have* must be present in the final solution. They represent the top priority and they are critical for the project's success.

Should Have (S): *Should Have* requirements represent a medium priority, as they are important and have some degree of urgency. However, the project's success is not entirely dependent on them.

Could Have (C): *Could Have* requirements are great additions to the project, but their implementation is not urgent. They represent a low priority.

Won't Have (W): This set of requirements represent a minimal level of priority. *Won't Have* requirements will not be implemented, although it could be developed in future iterations.

The functional requirements and their respective priorities can be found in Table 5.1.

ID	Functional Requirement	Priority
	Source Handling	
FR1	Import JSON files.	(M)
FR2	Import CSV files.	(S)
FR3	Import XML files.	(C)
FR4	Extract sensor file schema.	(M)
FR5	Extract database schema.	(M)
	Matching Process	
FR6	Automatically select the set of tables in the database to match a file.	(W)
FR7	Match a file to a pre-selected group of tables.	(M)
FR8	Compute linguistic similarities.	(M)
FR9	Compute semantic similarities.	(M)
FR10	Compute data type similarities.	(S)
FR11	Compute similarities through external sources of knowledge.	(S)
FR12	Compute phonetic similarities.	(C)
FR13	Compute structural similarities.	(C)
	Result Combination and Verdict	
FR14	Combine the results from the matchers.	(M)
FR15	Extract $1:1$ mappings.	(M)
FR16	Extract $n:1$ mappings.	(S)
FR17	Extract $n:m$ mappings.	(C)
FR18	Export mappings.	(M)
	Statistics	
FR19	Show total run-time.	(M)
FR20	Show run-time for each task.	(S)
FR21	Show precision, recall and f1-score percentages.	(M)

Table 5.1: Functional requirements

5.3 Non-Functional Requirements

Non-functional requirements are quality attributes that complement the functionality of the system, specifying how the system must behave in certain scenarios. They indicate how well the system satisfies the stakeholders' needs [138].

The following subsections detail the non-functional requirements for this project. Similarly to functional requirements, one can rank non-functional requirements. Their priorities can be found in Table 5.2

ID	Non-Functional Requirement	Priority
NFR1	Maintainability	(M)
NFR2	Performance	(S)

Table 5.2: Non-functional requirements

5.3.1 Maintainability

Maintenance is a reality for all systems. Either because new requirements need to be met, or due to the necessity of replacing a faulty piece. Good maintenance is key to a product's longevity, so, the easier it is to maintain it, the better. Maintainability is a way of characterizing how easy is to perform maintenance on a system component. It can be expressed into more specific requirements such as modularity and modifiability, which have a notable applicability in this project [80].

Modularity: Modularity translates into the ability of dividing the system into smaller components in a way that they can operate independently, completing part of a bigger task. For this specific project, modularity is particularly relevant, not only to facilitate the system building and its reconfiguration, but also to help testing subsystems individually.

Modifiability: Modifiability is the capacity of a system to be modified without introducing bugs or reducing the software quality. Since this project is the first iteration of a data model conversion system, it is important to establish good foundations to support future changes and additions.

Maintanability scenarios can be found in Tables 5.3 and 5.4.

Source of Stimulus	Developer
Stimulus	Developer wants to add a new module to the system.
Environment	Development
Artifact	Source code.
Response	The addition does not cause any side effects on the system.
Response Measure	The addition takes less than twenty minutes.

Table 5.3: Maintainability scenario (addition)

5.3.2 Performance

Performance quantifies the amount of resources used to execute the program under various circumstances [138]. Resources may include time, memory, CPU usage etc. It was defined

Source of Stimulus	Developer
Stimulus	Developer wants to replace a module.
Environment	Development
Artifact	Component's source code.
Response	The substitution does not cause any side effects on the system.
Response Measure	The substitution takes less than five minutes.

Table 5.4: Maintainability scenario (substitution)

that the performance of the developed system would be measured as the run-time. The defined threshold for the execution time is given by the time an average Ubiwhere's employee takes to complete the manual integration. The performance scenario is showed in Table 5.5.

Source of Stimulus	Airflow DAG.
Stimulus	New source import.
Environment	Normal conditions.
Artifact	System
Response	The system returns the mapping.
Response Measure	The system takes less than five minutes to return the mapping.

Table 5.5: Performance scenario

5.4 Restrictions

Ubiwhere imposed restrictions with respect to software and data during development and testing. Any software used for this project had to be open-source. In addition, all real data used to test the solution, needed to be open-source or already Ubiwhere's property, instead. Such restrictions were purely due to business reasons, since Ubiwhere wanted to reduce the investment in development as much as possible. Furthermore, the company wanted to avoid relying on vendors' licences and policies.

5.5 Architecture

Software architecture is a continuous development process throughout the project that aims at finding the best system organization in order to meet the requirements established. It should also describe the relationships between the different parts.

As mentioned several times across this document, the purpose of the automatic matching system is to replace a traditional manual conversion. The manual conversion is part of UBP's Airflow component, which is responsible for importing and mapping the sources, as explained in section 2.1 and depicted in Figure 2.1. Figures 5.1 and 5.2 denote the differences between the two approaches. In the classic way, each DAG has an import function, a function to map the arriving files to UBP's data models, and a sender, which forwards the data to the next phase. On the other hand, the new approach resorts to the same mapping system for every DAG, contributing to a quicker and easier integration of new sources, represented by new DAGs. It is important to point out that in the new configuration the automatic converter runs just once for every new file source, so that, if

a new file from the same source arrives, the mapping previously determined and stored in cache is returned immediately, without the necessity of matching the data models again.

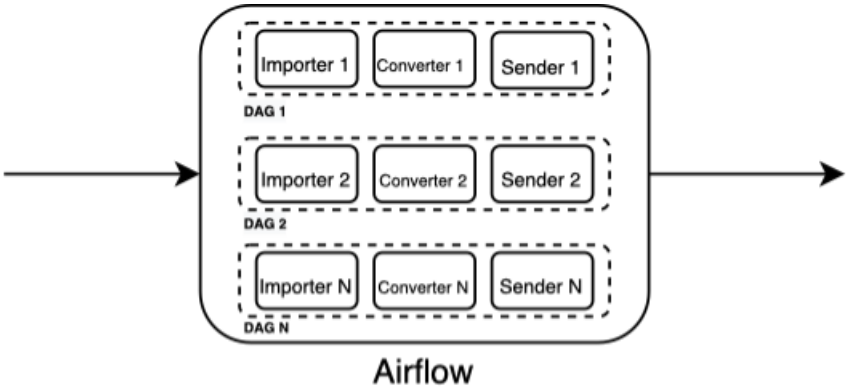

Figure 5.1: Traditional Airflow configuration

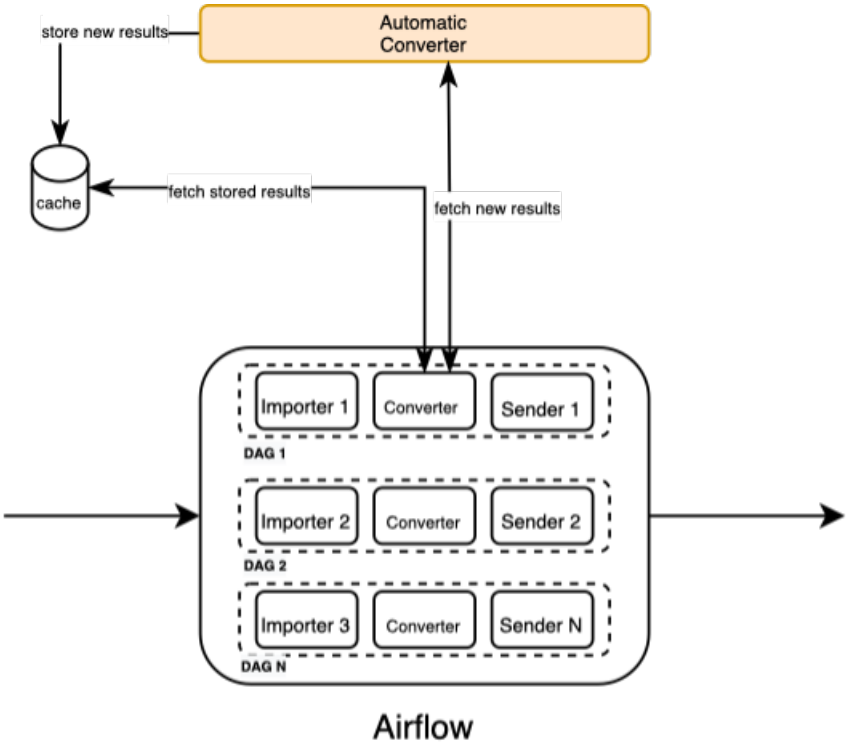

Figure 5.2: New Airflow configuration proposal

General architecture can be seen in Figure 5.3. Arrows represent data flows. An overview of each module's inputs and outputs is pictured in Figure 5.4. Given the complexity and requirements of the project, the automatic converter system was built of several modules. Each module is in charge of a specific phase of the pipeline. The modules are *Schema Module*, *Normalization Module*, *Candidate Selection Module*, *Parallel Matching Module*, *Mapping Module* and *Statistics Module*. Further details on them will be given in the following subsections.

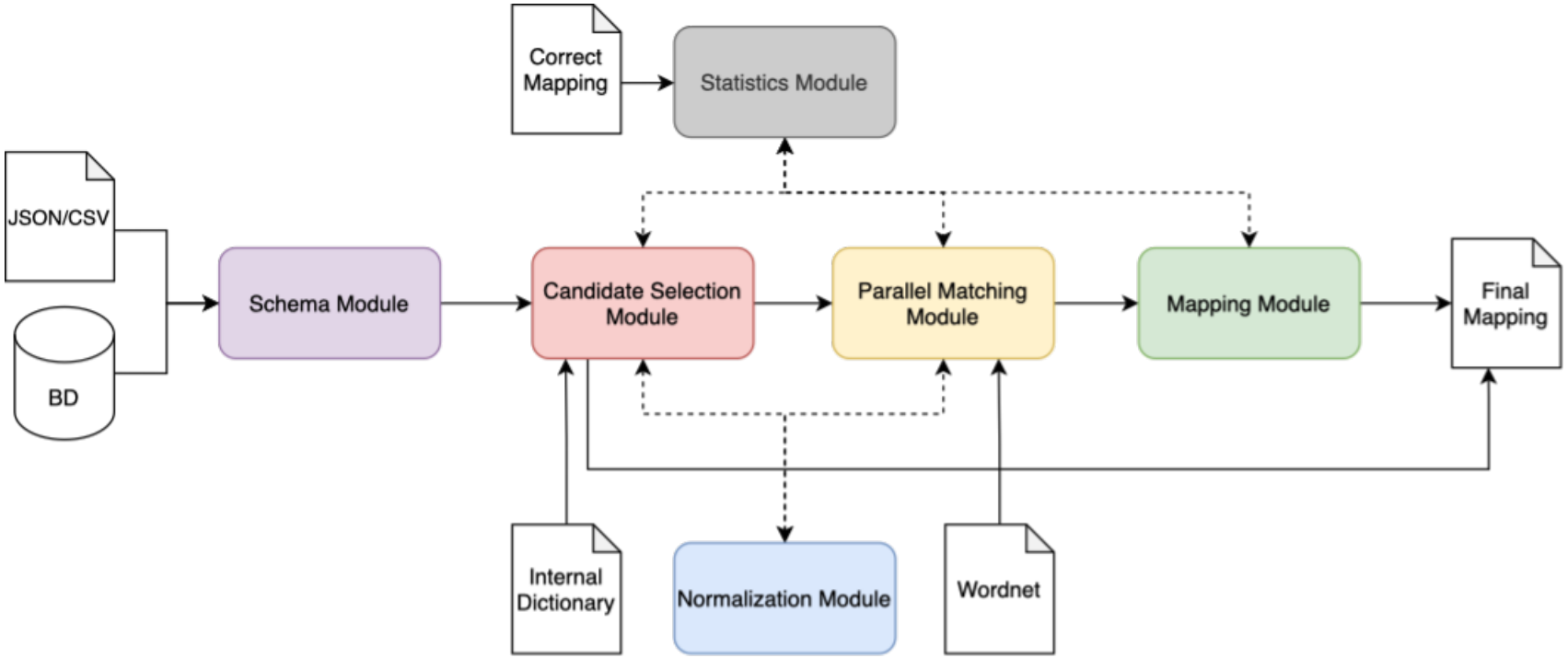

Figure 5.3: General architecture of the automatic converter

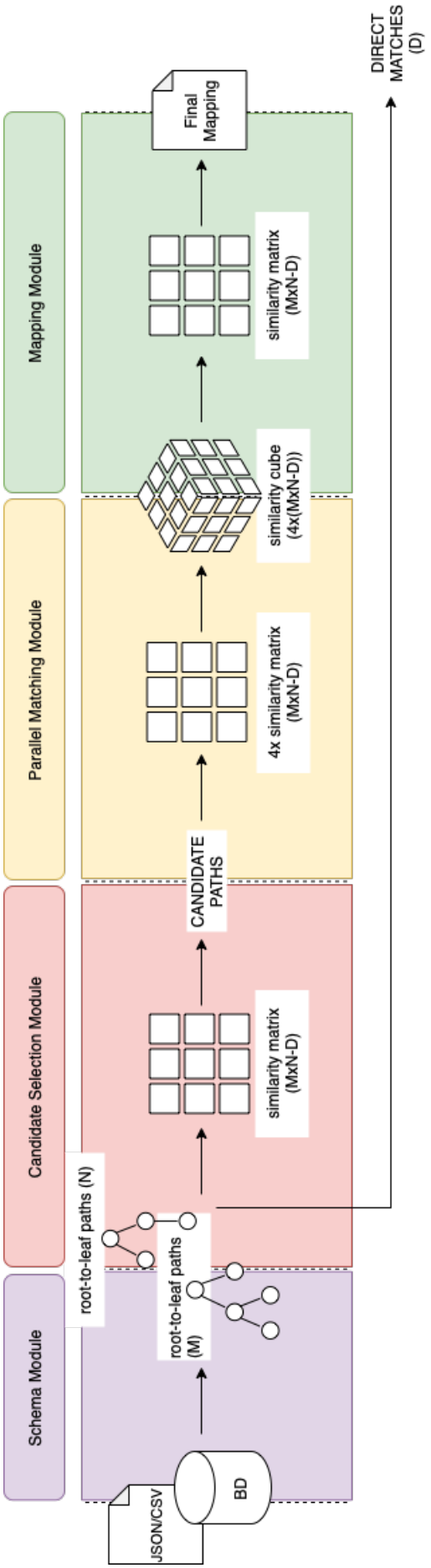

Figure 5.4: Inputs and outputs overview

5.5.1 Schema Module

Schema Module is the first module of the pipeline. It is responsible for handling the imported files, extracting the schemas and converting them into a common internal tree format, which allows easy access to attributes and grants the permanence of each source's structure. This module's architecture can be fount in Figure 5.5 and it is composed of three smaller components, each one responsible for treating a different input format. *Table Extractor* picks the tables in the database, *Attribute Extractors* and *Column Extractor* obtain the attribute names, *Data Types Extractors* verify the data type of a given attribute, *Relationship Extractors* check if an attribute has descendants and *Graph Builders* build the graph, encoding all the previously obtained information into nodes and edges. This module receives raw files from city sensors as input and outputs a common graph structured representation of these sources.

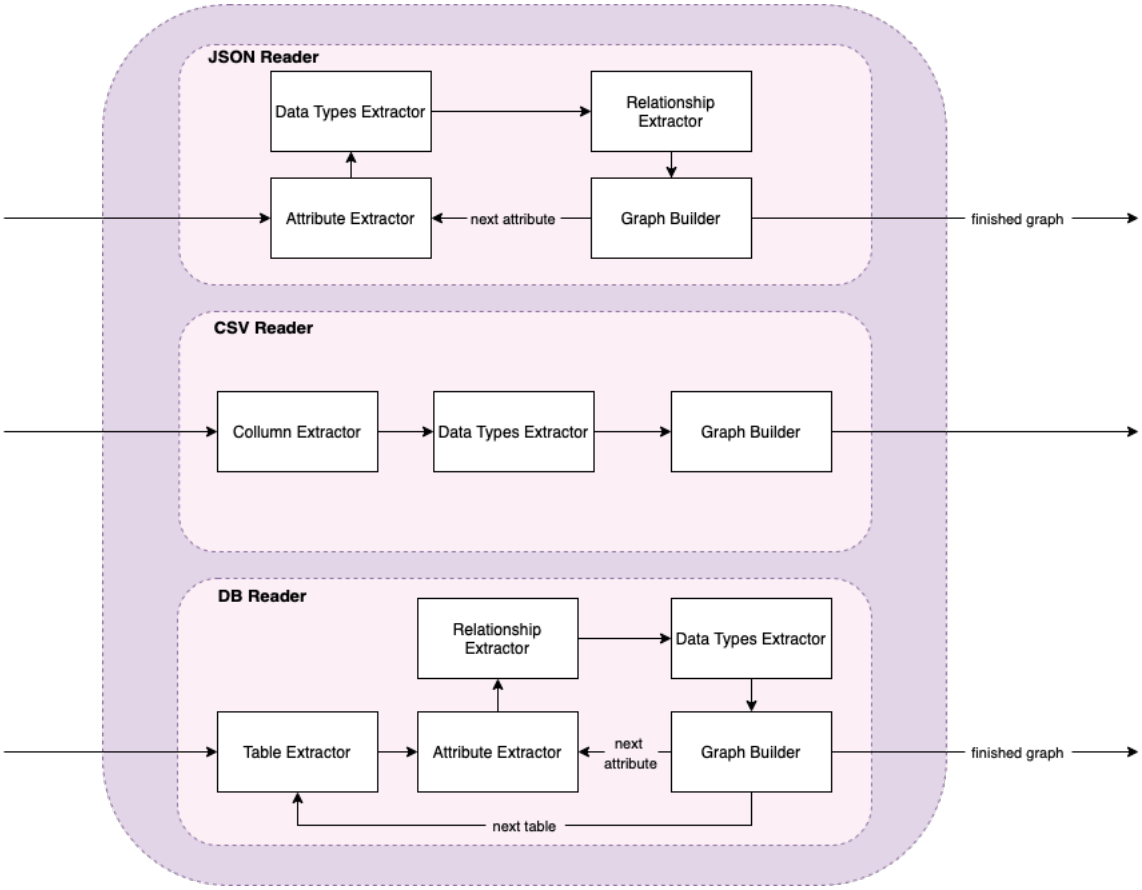

Figure 5.5: *Schema Module* architecture

5.5.2 Normalization Module

Normalization Module is an complementary component composed of several normalization functions, pictured in Figure 5.6, useful in the several stages of the system. These functions treat the attributes input, returning the normalized strings, before other modules perform operations on them.

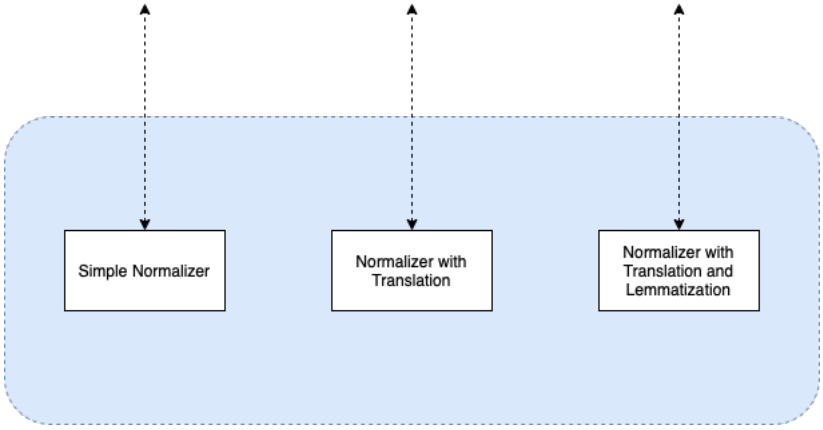

Figure 5.6: *Normalization Module* architecture

5.5.3 Candidate Selection Module

After the parsing of the sources, *Candidate Selection Module* extracts all paths, from the root node to the leaves of the graphs, and computes a similarity based on the embeddings of the paths. From these similarities, the most similar pairs are appointed as possible match candidates. An internal dictionary formed with specific scientific terminologies also extracts direct matches. The module returns direct matches and a list of candidate pairs, that are going to be analyzed in the following stages. *Candidate Selection Module* architecture is schematized in Figure 5.7.

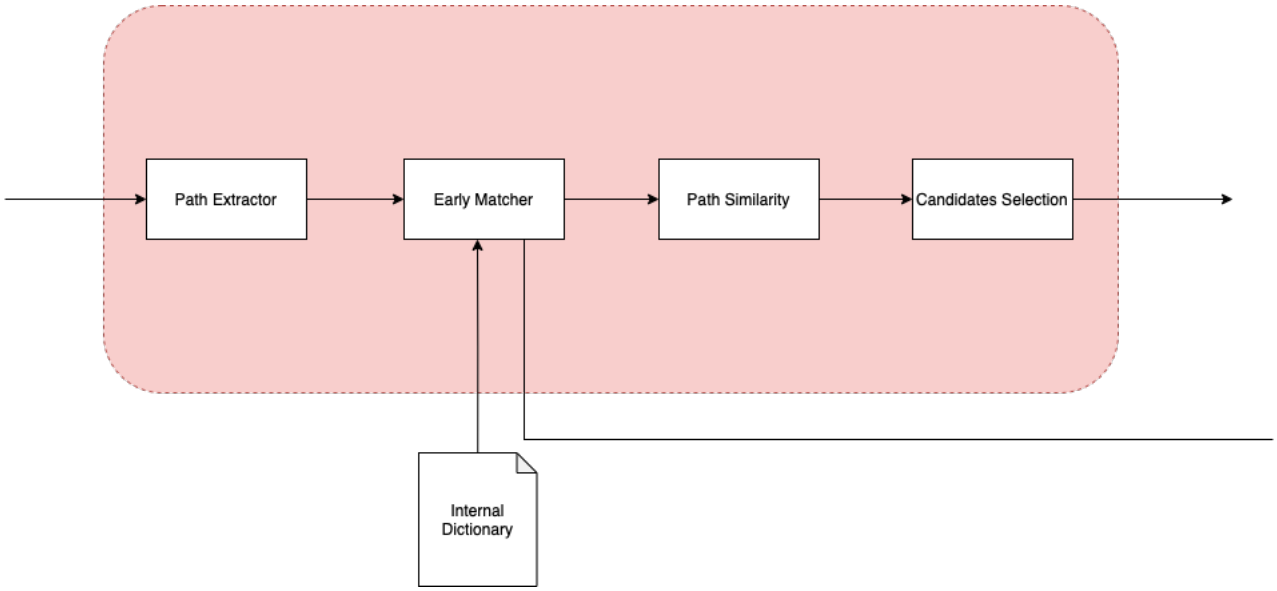

Figure 5.7: *Candidate Selection Module* architecture

5.5.4 Parallel Matching Module

Seen in Figure 5.8, *Parallel Matching Module* is composed of smaller modules that compute different similarities in parallel, exploring different features of the schemas. This module includes a string matcher, a token matcher, a matcher based on external information and a type matcher, but the idea is that new matchers can easily be added to the set of existing

ones. Candidate paths determined before are loaded and a similarity cube, shaped from every matcher's similarity matrices, is sent to the next phase.

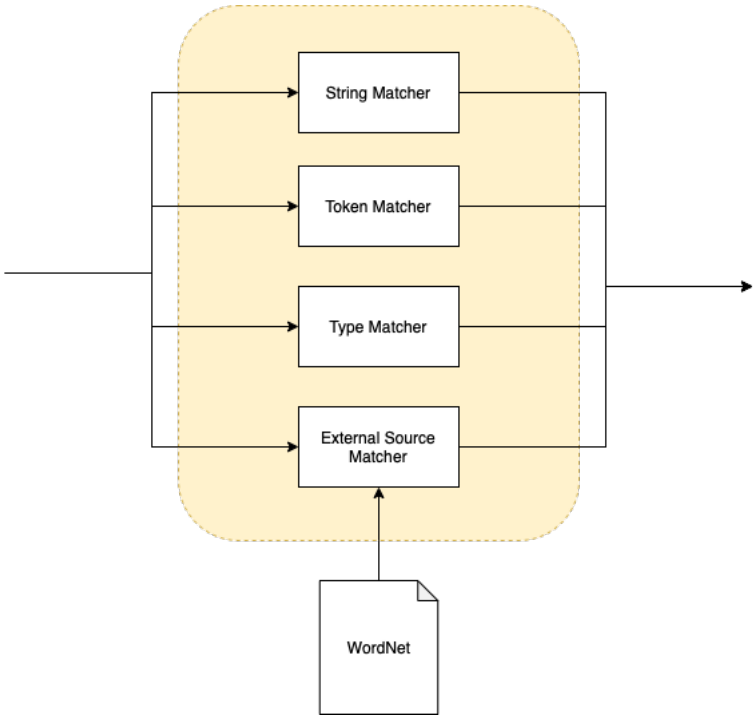

Figure 5.8: *Parallel Matching Module* architecture

5.5.5 Mapping Module

Mapping Module combines the results of the parallel matchers in a single matrix, gives the verdict and returns the final mappings between attributes of the new source and UBP's database attributes. The module diagram is shown in Figure 5.9.

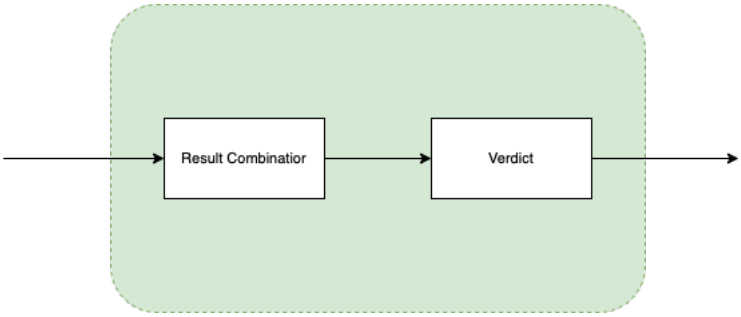

Figure 5.9: *Mapping Module* architecture

5.5.6 Statistics Module

Statistics Module is an auxiliary module whose job is to calculate statistics and display plots. This module can be used throughout the system to easily analyze the performance the different modules and help in the validation of the system. The architecture is represented in Figure 5.10.

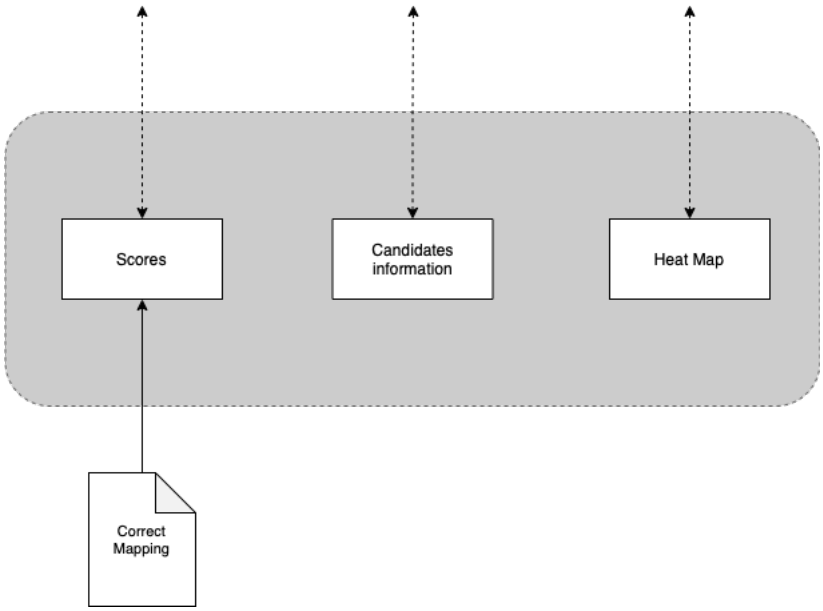

Figure 5.10: *Statistics Module* architecture

5.6 Technologies

The technologies chosen for the development of this project were primarily based on the ones already used in UBP and by Ubiwhere, in general, because it would facilitate the integration of the developed system and Ubiwhere's employees could perform changes without major difficulties. Each technology had to meet the set of features needed to implement the project's list of functionalities. Besides that, the popularity of the technology and the documentation available, heavy influenced the choices. Finally, despite not being many, restrictions were taken into account too, so that all technologies adopted were open-source.

Python was the language chosen for the development of the project, mainly because it was the language in which the UBP was coded, but also because it is familiar to most developers, specially at Ubiwhere. Regarding libraries, Python offers a variety of well-stocked tools, ideal for expedite the project development.

5.6.1 Schema Module

For the *Schema Module*, one imperative library is the NetworkX[1] library, which provides a set of functions to easily build and represent graph-like structures. It was released in 2008 [75], but it has since been updated multiple times, being one of the most, if not the most, well documented open-source library for building graphs. Thus it was utilized to convert the sources to be matched into tree structures. Among other libraries chosen for this module are *psycopg2*[2], the PostgreSQL database adapter used to query tables' information; *json*[3] and *csv*[4], two libraries for handling JSON and CSV files, respectively; *datetime*[5] and *dateutil*[6], used to parse date and time attribute data types.

[1] https://networkx.org/documentation/stable/index.html
[2] https://www.psycopg.org
[3] https://docs.python.org/3/library/json.html
[4] https://docs.python.org/3/library/csv.html
[5] https://docs.python.org/3/library/datetime.html
[6] https://dateutil.readthedocs.io/en/stable/

5.6.2 Normalization Module

Normalization Module, as explained before, is a module with a set of functions to normalize strings. Therefore, the modules used in this component are *unicodedata*[7], to help treat special characters; *re*[8] to build regular expressions that remove unecessary parts of strings; NLTK[9], a kit with lots of tools to process text, including stemmers, lemmatizers, tokenizers etc.; *googletrans*[10], a free version of Google Translate API. All these libraries are free to use, well documented, well maintained, with the exception of *googletrans*, and NLTK even has an active discussion forum.

5.6.3 Candidate Selection Module

Just like *Schema Module*, *Candidate Selection Module* uses NetworkX to be able to process the schemas in a tree format and *json* to interpret the internal dictionary. To build the similarity matrix, *numpy*[11] library was the choice, since it is open-source and well known for its performance when manipulating large arrays. For calculating the similarities, the choices were SentenceTransformers[12], a framework for text and image embeddings, which offers a large variety of free pre-trained models; *scikit-learn*[13], a large open-source machine learning library that implements lots of functions including metrics like the cosine similarity used in this module.

5.6.4 Parallel Matching Module

Parallel Matching Module runs several matchers at the same time. For that, it was decided that *multiprocessing*[14] library was preferred over *threading*, because it allows the functions to run concurrently, bypassing the Global Interpreter Lock (GIL) [108]. Once again, to build similarity matrices, *numpy* was chosen and NLTK was used as well but this time to access WordNet. To compute edit distances *pyjarowinkler*[15] and *Levenshtein*[16] extensions were chosen.

5.6.5 Mapping Module

Mapping Module combines all the similarity matrices, so the only library needed for this module is *numpy*.

[7] https://docs.python.org/3/library/unicodedata.html
[8] https://docs.python.org/3/library/re.html
[9] https://www.nltk.org
[10] https://pypi.org/project/googletrans/
[11] https://numpy.org
[12] https://www.sbert.net
[13] https://scikit-learn.org/stable/index.html
[14] https://docs.python.org/3/library/multiprocessing.html
[15] https://pypi.org/project/pyjarowinkler/
[16] https://pypi.org/project/Levenshtein/

5.6.6 Statistics Module

Statistics Module utilizes *json* to read mapping files and *numpy* to work with the similarity matrices. Regarding visualization, *pydot*[17] and *Graphviz*[18] help building the graph structure of the sources to be matched. Additionally, the libraries *seaborn*[19] and *matplotlib*[20] were employed to help plotting the similarity matrices in a more readable and pleasant way.

5.7 Conclusion

This chapter enumerated the functional and non-functional requirements agreed with the stakeholders, ranked by levels of priority, according to their importance for the project. They dictate what is going to be done and, together with the restrictions, they help shaping the architecture. Lastly, the general architecture was presented. All architectural and technological decisions were described and duly justified.

Next chapter guides the reader through the process of implementation, showing step-by-step how the different algorithms accomplish their purpose.

[17] https://pypi.org/project/pydot/
[18] https://graphviz.org
[19] https://seaborn.pydata.org
[20] https://matplotlib.org

This page is intentionally left blank.

Chapter 6

Implementation

Implementation constitutes the development of a solution that materializes the architecture and, subsequently, meets the requirements previously gathered and specified.

This chapter presents the steps followed during the implementation of this project, accompanied by an explanation of the modules' technicalities and algorithms at a lower level.

6.1 Environment

Before writing any code it is unavoidable having to set up the development environment. As mentioned already in chapter 4, GitLab was the organizational platform and version control tool adopted. A new repository was created just to accommodate this project. During the implementation, updates would be pushed to a *development* branch.

Together with the cloned repository, a virtual environment was configured on the development computer. A virtual environment helps isolating different projects' dependencies and it can be conveniently created using python libraries like *virtualenv*[1] or *venv*[2]. On this environment, all the basic dependencies to run UBP's Airflow importers were installed, along with the libraries needed for the system that was being implemented.

The modules were organized in distinct python scripts. Moreover, inside the project folder there is even a separate folder with test datasets and their respective mappings. The database was reproduced using a dump of the UBP's database, supplied by Ubiwhere.

It was decided that it made sense developing this project in a separate environment from the rest of the UBP to accelerate the set up of the environment and simplify the testing. On top of that, detaching the system development from the already working UBP, promoted the creation of a modular solution that does not depend too much on the application context.

6.2 Functionalities

The majority of the functionalities were implemented, as it can be seen in Table 6.1, which condenses the functional requirements and their status at the time of this report.

[1] https://virtualenv.pypa.io/en/latest/
[2] https://docs.python.org/3/library/venv.html

ID	Functional Requirement	Status
	Source Handling	
FR1	Import JSON files.	Complete
FR2	Import CSV files.	Complete
FR3	Import XML files.	Lacking
FR4	Extract sensor file schema.	Complete
FR5	Extract database schema.	Complete
	Matching Process	
FR6	Automatically select the set of tables in the database to match a file.	Lacking
FR7	Match a file to a pre-selected group of tables.	Complete
FR8	Compute linguistic similarities.	Complete
FR9	Compute semantic similarities.	Complete
FR10	Compute data type similarities.	Complete
FR11	Compute similarities through external sources of knowledge.	Complete
FR12	Compute phonetic similarities.	Lacking
FR13	Compute structural similarities.	Lacking
	Result Combination and Verdict	
FR14	Combine the results from the matchers.	Complete
FR15	Extract $1:1$ mappings.	Complete
FR16	Extract $n:1$ mappings.	Complete
FR17	Extract $n:m$ mappings.	Lacking
FR18	Export mappings.	Complete
	Statistics	
FR19	Show total run-time.	Complete
FR20	Show run-time for each task.	Complete
FR21	Show precision, recall and f1-score percentages.	Complete

Table 6.1: Status of functional requirements

Given the exploratory nature of the development, very much based on the testing and analysis of the results, some algorithms suffered changes over the course of this project. Although there is still room for improvements, it is important to emphasize that this version of the system carries the basic functionalities of a system of this kind , represented by high priority functional requirements (Table 5.1). These functionalities are enumerated and clarified in the next subsections.

6.2.1 Schema Module

Schema Module is the first module of the pipeline and its functions correspond to the first stages of the matching process. Three smaller components form this module and each one has the role of dealing with one file format. They are *JSON Reader*, *CSV Reader* and *Database Reader*. The returned graphs display a structure similar to the one in Figure 6.4.

JSON Reader

Most sources provide their data in JSON format. JSON is a semi-structured file format, which means it does have not follow a strict schema. Its flexibility made JSON a popular format for exchanging information in APIs and web applications over the last decade [128]. Hence, being able to handle JSON files was indispensable for this project.

After fetching the source file, to extract its structure a function *graphBuilder* is called (see Algorithm 1). This function recursively builds the graph that represents the structure of the file. It starts by checking if the value of an attribute is a dictionary or a list. If it is a list, it means that either the value is an instance 6.1 or another structure has been reached 6.2.

```
{
    "attribute_A": ["value1", "value2", "valueN"]
}
```

Listing 6.1: JSON instance inside a list

```
{
    "attribute_A":
        [
            {
                "attribute_Aa": "value1",
                "attribute_Ab": "value1",
            },
            {
                "attribute_Aa": "value2",
                "attribute_Ab": "value2",
            },
        ]
}
```

Listing 6.2: JSON structure inside a list

On the contrary, if a dictionary is spotted, it is a sign that the end attribute has not been reached yet 6.3, so the new node is added and the function is called again to crawl over the value of that attribute.

```
{
    "attribute_A": {
        "attribute_B": {
            attribute_C: "value1"
        }
    }
}
```

Listing 6.3: JSON attribute of and attribute

Together with the attribute name, a property stating the attribute's type is added. Sometimes, various data types come encapsulated in strings, which hinders the correct correspondence to UBP's database columns. For that, a group of functions specializes in finding the real data type of that attribute. These functions evaluate whether the value is really a string or if can be represented as an integer, decimal or even date.

Algorithm 1 JSON reader

```
function jsonReader(source_name)
    root ← source_name
    data ← fetchSource(root)
    G ← DAG(root)
    G ← graphBuilder(G, root, data)
    return G
end function

function graphBuilder(G, parent, data)
    if data is a list then
        graphBuilder(G, root, data[0])
    else if data is a dictionary then
        for k in data do
            type ← checkType(data[k])
            G ← addNode(G, k, type)
            G ← addEdge(G, parent, k)
            graphBuilder(G, k, data[k])
        end for
    end if
    return G
end function
```

CSV Reader

This is the simplest component of the *Schema Module*. The file is opened, the attribute names present in the header are extracted and a sample of the instances is analyzed to verify what is the data type. To ascertain the types, the same set of functions used in *JSON Reader* is called. Algorithm 2 shows the pseudo-code that builds the graph, whose nodes are named as column names and data types are stored in node too.

Database Reader

Algorithms 3 and 4 show the steps to build the database schema. In *dbReader* the connection with the database is established and the graph is initialized. After that, all the table names are queried and stored (Listing 6.4). This list is important to keep track of what tables' structures have already been analyzed. Function *buildTable* is responsible for building the schema. For a given table, it queries the column names and all foreign keys with the help of *queryTableInfo*. The queries used for this can be found in Listings 6.5 and 6.6. Then, the column nodes are added to the graph and in case of existing foreign keys, the program checks if the referenced tables have already been traversed and, consequently, if their structure is already part of the schema. If that is the case, the program simply checks what is the column which is being referenced, by calling *checkRefCol*, and adds a new edge to it. If not, it builds the referenced table before adding the new edge.

Algorithm 2 CSV reader

function CSVREADER($source\_name$)
  $root \leftarrow source\_name$
  $data \leftarrow fetchSource(root)$
  $G \leftarrow DAG(root)$
  $G \leftarrow$ GRAPHBUILDER$(G, root, data)$
  return G
end function

function GRAPHBUILDER($G, data$)
  $header \leftarrow readLine(data)$
  $count \leftarrow 0$
  for c in $header$ **do**
    $type \leftarrow checkType(readLine(data)[count])$
    $G \leftarrow addNode(G, c, type)$
    $G \leftarrow addEdge(G, root, c)$
    $count \leftarrow count + 1$
  end for
  return G
end function

```
SELECT table_name
FROM information_schema.tables
WHERE table_schema = '<schema_name>';
```

Listing 6.4: Query to get all table names

```
SELECT *
FROM information_schema.columns
WHERE table_schema = '<schema_name>' AND table_name = '<table_name>';
```

Listing 6.5: Query to get columns' information

```
SELECT
    from_table.table_name AS from_table,
    from_column.column_name AS from_column,
    to_table.table_name AS to_table,
    to_table.column_name AS to_column
FROM information_schema.table_constraints AS from_table
    JOIN information_schema.constraint_column_usage AS to_table
        ON to_table.constraint_name = from_table.constraint_name AND
    to_table.table_schema = from_table.table_schema
    JOIN information_schema.key_column_usage AS from_column
        ON from_column.constraint_name = from_table.constraint_name  AND
    from_column.table_schema = from_table.table_schema
WHERE from_table.constraint_type = 'FOREIGN KEY' AND from_table.
    table_name='<table_name>' AND from_table.table_schema = '<schema_name
    >';
```

Listing 6.6: Query to get foreign keys

Algorithm 3 Database reader

```
function dbReader
    G ← DAG()
    try
        connection ← dbConnection()
        tables ← queryTableNames()
        for table in tables do
            G, tables ← buildTable(G, tables, table)
        end for
    catch Exception
    finally
        if connection then
            closeConnection(connection)
            return G
        end if
    end try
end function


function buildTable(G, tables, table)
    columns, fks ← queryTableInfo(table)
    for c in columns do
        type ← c[1]                    ▷ c[0] is the column's name and c[1] its data type
        G ← addNode(G, c[0], type)
        G ← addEdge(G, table, c[0])
        for fk in fks do
            if fk[1] = c[0] then       ▷ f[1] is the name of the origin table and f[2] is the
referenced table
                ref_table ← fk[2]
                if ref_table in tables then
                    buildTable(G, tables, ref_table)
                end if
                ref_col ← checkRefCol(G, fk[3], ref_table)   ▷ fk[3] is the referenced
column
                G ← addEdge(G, ref_col, c[0])
            end if
        end for
    end for
    tables.remove(table)
    return G, tables
end function
```

Algorithm 4 Database reader (auxiliary functions)

```
function queryTableInfo(table)
    cols ← queryColumns()
    fks ← queryFKs()
    columns ←[ ]
    for c in cols do                    ▷ c[3] is the column name and c[27] its data type
        columns.insert([c[3], c[27]])
    end for
    return columns
end function

function checkRefCol(G, item, root)
    for node in G.successors(root) do
        if item in node then
            return node
        end if
    end for
end function
```

6.2.2 Normalization Module

Normalization Module comprises three normalization functions. The *simpleNormalization*, Algorithm 5, is actually the basis of the other functions. It starts by removing the accents. The accents can mislead a matcher especially when working with Romance languages. Then, special characters, such as punctuation and digits, are removed, followed by the replacement of underscores (used in Snake case) and Camel cases by a white space. Afterwards, the word is converted to lowercase. It finishes by striping the extra spaces that might be on the edges or in the middle of the string. As explained in the last chapter, libraries like *re* and *unicodedata* support this process. Some examples of inputs and outputs can be found bellow in Table 6.2.

Input	Output
parkingSpot	*parking spot*
sCheDúle!	*schedule*
*Address*01	*address*
Street Name	*street name*

Table 6.2: *simpleNormalization* input and output examples

Algorithm 5 Simple normalization

```
function simpleNormalization(word)
    norm_word ← removeAccents(word)
    norm_word ← removeSpecialChars(norm_word)
    norm_word ← replaceCamelSnake(norm_word, ' ')
    norm_word ← toLower(norm_word)
    norm_word ← stripWord(norm_word)
    return norm_word
end function
```

The second algorithm is a *simpleNormalization* at its core but instead it identifies in which

language the word/expression is written and, if it is not in English, it tries to translate the word/expression, as it can be seen in Algorithm 6.

Algorithm 6 Normalization with translation

```
function transNormalization(word)
    norm_word ← simpleNormalization(word)
    lang ← getLanguage(norm_word)
    if lang != 'en' then
        norm_word ← translateWord(norm_word, 'en')
    end if
    return norm_word
end function
```

Finally, the third function, does everything the others do, but also lemmatizes the word. Different verb conjugations like *feet*, *foot* are graphically different, but have the same lemma, *foot*, which may help identifying similar concepts. Once again, the algorithm can be found in Algorithm 7 and examples of inputs and outputs can be observed in Table 6.3.

Algorithm 7 Normalization with translation and lemmatization

```
function transLemNormalization(word)
    norm_word ← transNormalization(norm_word)
    norm_word ← split(norm_word, ' ')
    result ← ''
    count ← 0
    for w in norm_word do
        if count != 0 then
            result ← result+ ' '
        end if
        w ← lemmatize(w)
        result ← result + w
        count ← count + 1
    end for
    return result
end function
```

Input	Output
parkingSpots	*parking spot*
Ruas_?	*street*
Corpora01	*corpus*

Table 6.3: *transLemNormalization* input and output examples

6.2.3 Candidate Selection Module

Candidate Selection Module selects obvious matches and the pairs that have higher chances of being matches. Its steps are shown in Algorithm 8. The first step is to get all the paths from the root to the leaves (Algorithms 9 and 10). This way, it is possible to encode all the information of each branch, represented by the leaf attribute.

Earlier matches are the direct correspondences between attributes (Algorithm 11). These matches do not constitute any kind of ambiguity and are mostly associated with certain

acronyms that represent specific measures, such as concentrations and air quality indices. These matches are kept in an internal dictionary (Listing 6.7).

```
{
    {
    "Chemistry":{
        "CO": ["carbon monoxide", "monoxido de carbono", "monoxyde de carbone"],
        "CO2": ["carbon dioxide", "dioxido de carbono", "anidrido carbonico", "dioxyde de carbone", "gaz carbonique", "anhydride carbonique"],
        "H2O": ["water", "agua", "agua", "eau"],
        "NO": ["nitric oxide", "oxido nitrico", "monoxido de nitrogenio", "monoxido de azoto", "oxyde azotique", "oxyde nitrique"],
        "NO2": ["nitrogen dioxide", "dioxido de nitrogenio", "dioxido de azoto", "dioxido de nitrogeno", "dioxyde dazote"]
    },
    "Ecology":{
        "AQI": ["air quality index", "indice de qualidade do ar"],
        "PM2.5": ["pm25", "Particulate matter 2.5"],
        "VOCs": ["volatile organic compounds", "composto organico volatil"]
    }
}
}
```

Listing 6.7: Sample of the internal dictionary

After the earlier matching, it is time to load the model and encode the paths (Algorithm 12). sBERT, or sentence BERT, is a network composed of two siamese BERT subnetworks. Its architecture allows greater and faster performance over traditional BERT networks. Besides the insights on embedders given in Chapter 3, additional information on sBERT and BERT can be found in [122] [89] [45] [46]. The library SentenceTransformers provides all the necessary tools as well as pre-trained models, from which *all-MiniLM-L6-v2* was selected. The official Hugging Face repository states that it was trained with over one billion training pairs from Reddit comments, Stack Exchange and Yahoo Answers posts, among other sources [66], which makes it suitable to be a general purpose model. Another report shows that the model offers one of the best relationships between sentence embedding performance and speed [121].

The similarity matrix is then filled with all pairs' cosine similarity scores. To determine what is a pair and what is not, the methodology HADAPT, detailed in Chapter 3. To the maximum values of each row and column is assigned the value one. For the positions housing a row and a column maximums at the same time, the value two is assigned (Algorithm 13). The hierarchy in candidates determine if more or less weight should be given to a candidate pair in future measurements. The effects of these operations are pictured by Figure 6.1.

6.2.4 Parallel Matching Module

This module has four matchers that run in parallel. Each one explores a different set of features that try to find commonalities between attributes. They are independent and execute in parallel, so that time is saved. However, with time saving efforts come some trade-offs, as the comparisons are only made between the end node attributes. For this project, only traditional techniques were employed, as modern machine learning ones require an abundance of training data not available at the moment. Furthermore, traditional

Algorithm 8 Candidates determination

```
function CANDIDATES(graph1, file_name, graph2, table_names, model_name)
    file_paths ← EXTRACTFILEPATHS(graph1, file_name)
    db_paths ← EXTRACTPATHSDB(graph2, table_names)
    earlier_matches, file_paths, db_paths ← ACRONYMMATCHES(file_paths, db_paths)
    model ← loadModel(model_n)
    sim_matrix ← PATHSIMILARITY(file_paths, db_paths, model))
    candidate_matrix ← CANDIDATESSELECTION(sim_matrix))
    return candidate_matrix
end function
```

Algorithm 9 File paths extraction

```
function EXTRACTFILEPATHS(graph, root)
    all_paths ← [ ]
    path ← [ ]
    all_paths ← BUILDFILEPATHS(root, path, all_paths)
    return all_paths
end function

function BUILDFILEPATHS(graph, root, path, all_paths)
    type ← getType(graph, root)
    path.insert([root, type])
    children ← getDescendants(graph, root)
    if not children and path not in all_paths then
        all_paths.insert(path)
    else
        for child in children do
            EXTRACTFILEPATHS(graph, child, path, all_paths)
        end for
    end if
    return all_paths
end function
```

Algorithm 10 Database paths extraction

```
function extractPathsDB(graph, table_names)
    all_paths ← [ ]
    path ← [ ]
    for root in table_names do
        all_paths ← buildPathsDB(root, path, all_paths)
    end for
    return all_paths
end function

function buildPathsDB(graph, root, path, all_paths)
    type ← getType(graph, root)
    path.insert([root, type])
    children ← getDescendants(graph, root)
    if not children and path not in all_paths then
        all_paths.insert(path)
    else                          ▷ If the attribute has children, means it is a foreign key
        for child in children do
            child_table ← tableOrigin(graph, child)
            all_paths ← buildPathsDB(graph, child, path, all_paths)
            for p in all_paths do
                if child_table in p and child in p then
                    path ← concat(path, p)
                    all_paths.insert(path)
                end if
            end for
        end for
    end if
    return all_paths
end function
```

Algorithm 11 Earlier matcher

```
function acronymMatches(file_paths, db_paths, internal_dict)
    data ← importFile(internal_dict)
    earlier_matches ← [ ]
    for category in data do
        for entry in data[category] do
            for f_path in file_paths do
                for db_path in db_paths do
                    file_paths, db_paths, earlier_matches ←
earlierMatcher(f_path, file_paths, db_path, db_paths, earlier_matches, entry)
                    file_paths, db_paths, earlier_matches ←
earlierMatcher(db_path, file_paths, f_path, db_paths, earlier_matches, entry)
                end for
            end for
        end for
    end for
    return earlier_matches
end function

function earlierMatcher(path1, file_paths, path2, db_paths, earlier_matches, entry)
    att1 ← path1[length(path1) − 1][0]
    if toLower(att1) = toLower(entry) then
        att ← path2[length(path1) − 1][0]
        if toLower(att) in data[category][entry] or toLower(att) = toLower(entry) then
▷ A normalization may be applied if desired
            earlier_matches.insert([path1, path2])
            file_paths.remove(path1)
            db_paths.remove(path2))
        end if
    end if
    return file_paths, db_paths, earlier_matches
end function
```

Algorithm 12 Path similarity

```
function pathSimilarity(paths1, paths2, model)
    matrix ← [ ]
    sent1 ←″
    sent2 ←″
    for r ← 0 to length(paths1) do
        p1 ← paths1[r]
        for att1 in p1 do
            if sent1 then
                sent1 ← sent1 + att1
            end if
            sent1 ← sent1+ ' '
        end for
        for c ← 0 to length(paths2) do
            p2 ← paths2[c]
            for att2 in p2 do
                if sent2 then
                    sent2 ← sent2 + att2
                end if
                sent2 ← sent2+ ' '
            end for
            matrix[r, c] ← similarity(sent1, sent2, model)
        end for
    end for
    return matrix
end function

function similarity(sent1, sent2, model)
    vec1 ← encode(sent1, model)
    vec2 ← encode(sent2, model)
    sim ← cosineSimilarity(vec1, vec2)
    return sim
end function
```

Algorithm 13 Candidates selection

```
function candidatesSelection(matrix)
    cand_matrix ← zerosMatrixShape(matrix)
    for row in getRows(matrix) do
        r, c ← getPosBestScore(row)
        matrix[r][c] ← matrix[r][c] + 1
    end for
    for col in getColumns(matrix) do
        r, c ← getPosBestScore(col)
        matrix[r][c] ← matrix[r][c] + 1
    end for
    return cand_matrix
end function
```

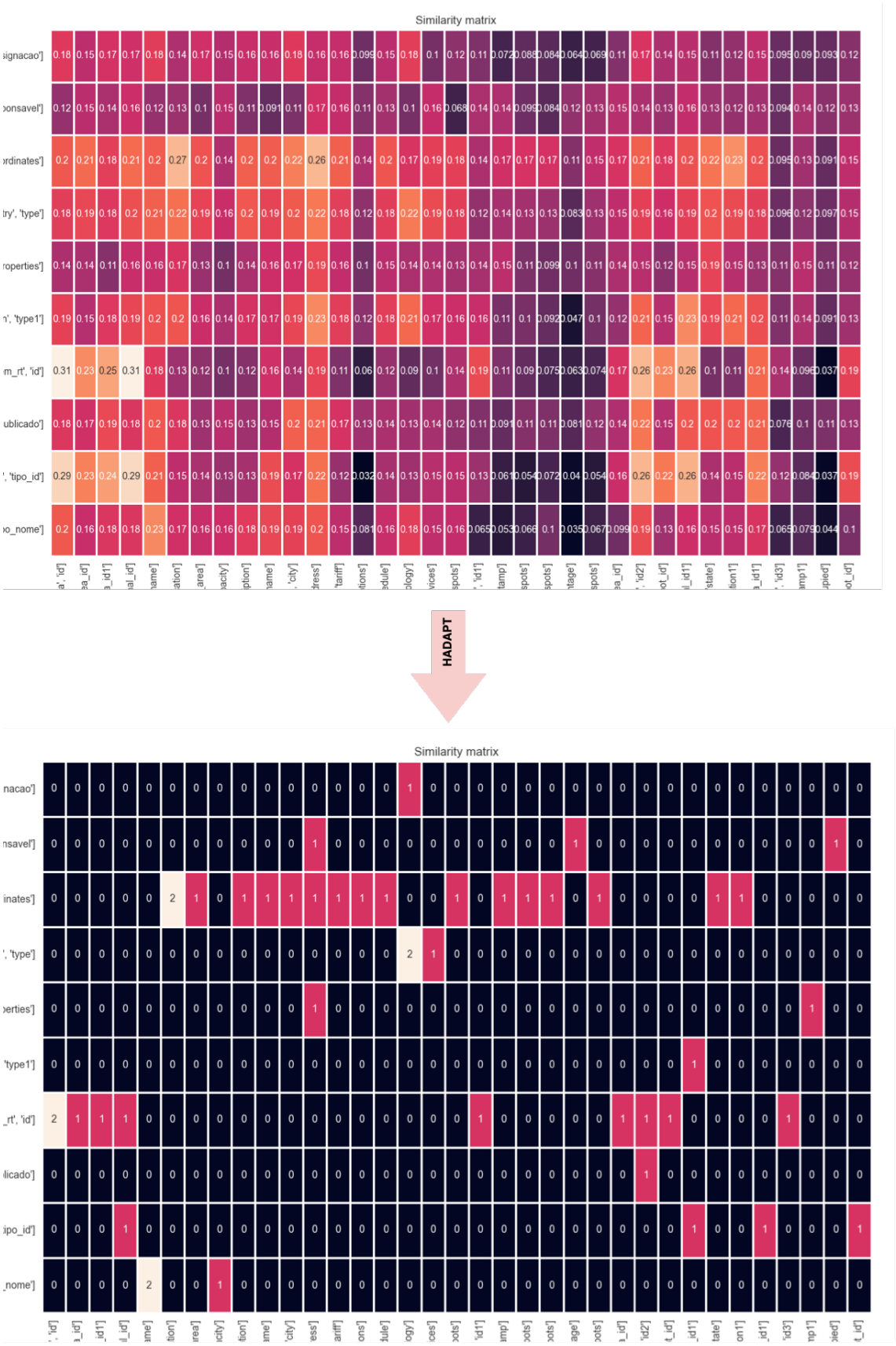

Figure 6.1: HADAPT method example

methods are reliable and continue to be implemented in most state of the art systems (as addressed in Section 3.5) .

String matching is analogous to almost every matching system. Comparing the two strings by their spelling is thus a fundamental operation. First, a similarity matrix is initialized to store the similarity values. This matrix's dimensions are the same as the previous module's, to facilitate the following stages of the pipeline, where the final maps are extracted. So, for every previously determined candidate pair, an edit distance is calculated and the other pairs remain zero. The matrix is then stored in a shared variable (Algorithm 14).

Algorithm 14 String matcher

```
function stringMatch(candidates, cand_matrix, shared_var)
    sim_matrix ← zerosMatrixShape(cand_matrix)
    for cand_list in candidates do        ▷ There are two candidate lists: one for good candidates and one for the best candidates
        for cand in cand_list do
            w1 ← cand[0][length(cand[0] − 1)][0]    ▷ the leaf attribute name of the first path
            w2 ← cand[1][length(cand[1] − 1)][0]    ▷ A normalization may be applied if desired
            max_l ← getMaxLength(w1, w2)
            s ← 1 − (levenshteinSim(w1, w2)/max_l)    ▷ Other measures, like Jaro-Winkler's, can be applied too
            pos ← cand[2]    ▷ checks the position of the pair on the matrix
            sim_matrix[pos] ← s
        end for
    end for
    shared_var.insert(sim_matrix)
end function
```

Another way of comparing two words is by breaking each word in smaller fixed sized grams and comparing them afterwards. It is exactly what the Algorithm 15 does. The elementary procedures are the same as the string matcher, but, instead of calculating an edit distance, it uses the Jaccard measure to return a score based on the common N-grams two words have.

Sometimes, comparing just the attributes' names is not sufficient. That is why it is important to gather as much information from as much sources as possible. Comparing the data types of two attributes should add a little bit more context to the comparison. This matcher implements the similarity measure described in [78] and explained in Chapter 3 (Algorithm 16). The hierarchy tree was adapted from W3C's data type hierarchy (Figure 6.3) and built upon estabilished correspondences such as [68], since the tree had to support multiple equivalent data types. The final hierarchy can be found in Figure 6.2. Notice that the two colors distinguish the Python data types and PostgreSQL's and boxes enclose equivalent data types. The fact that the tree is decoupled from the similarity measure itself, makes changing the hierarchy relatively easy.

Algorithm 15 Token matcher

```
function tokenMatch(candidates, cand_matrix, shared_var, n)
    sim_matrix ← zerosMatrixShape(cand_matrix)
    for cand_list in candidates do          ▷ There are two candidate lists: one for good candidates and one for the best candidates
        for cand in cand_list do
            w1 ← cand[0][length(cand[0] − 1)][0]      ▷ the leaf attribute name of the first path
            w2 ← cand[1][length(cand[1] − 1)][0]      ▷ A normalization may be applied if desired
            grams1 ← nGram(w1, n)
            grams2 ← CallnGramw2, n
            s ← jaccardSim(grams1, grams2)
            pos ← cand[2]                   ▷ checks the position of the pair on the matrix
            sim_matrix[pos] ← s
        end for
    end for
    shared_var.insert(sim_matrix)
end function


function nGram(word, n)
    grams ← [ ]
    for i ← 0 to length(word − n + 1)] do
        g ← word[i : i + n]
        grams.insert(gram)
    end for
    return grams
end function
```

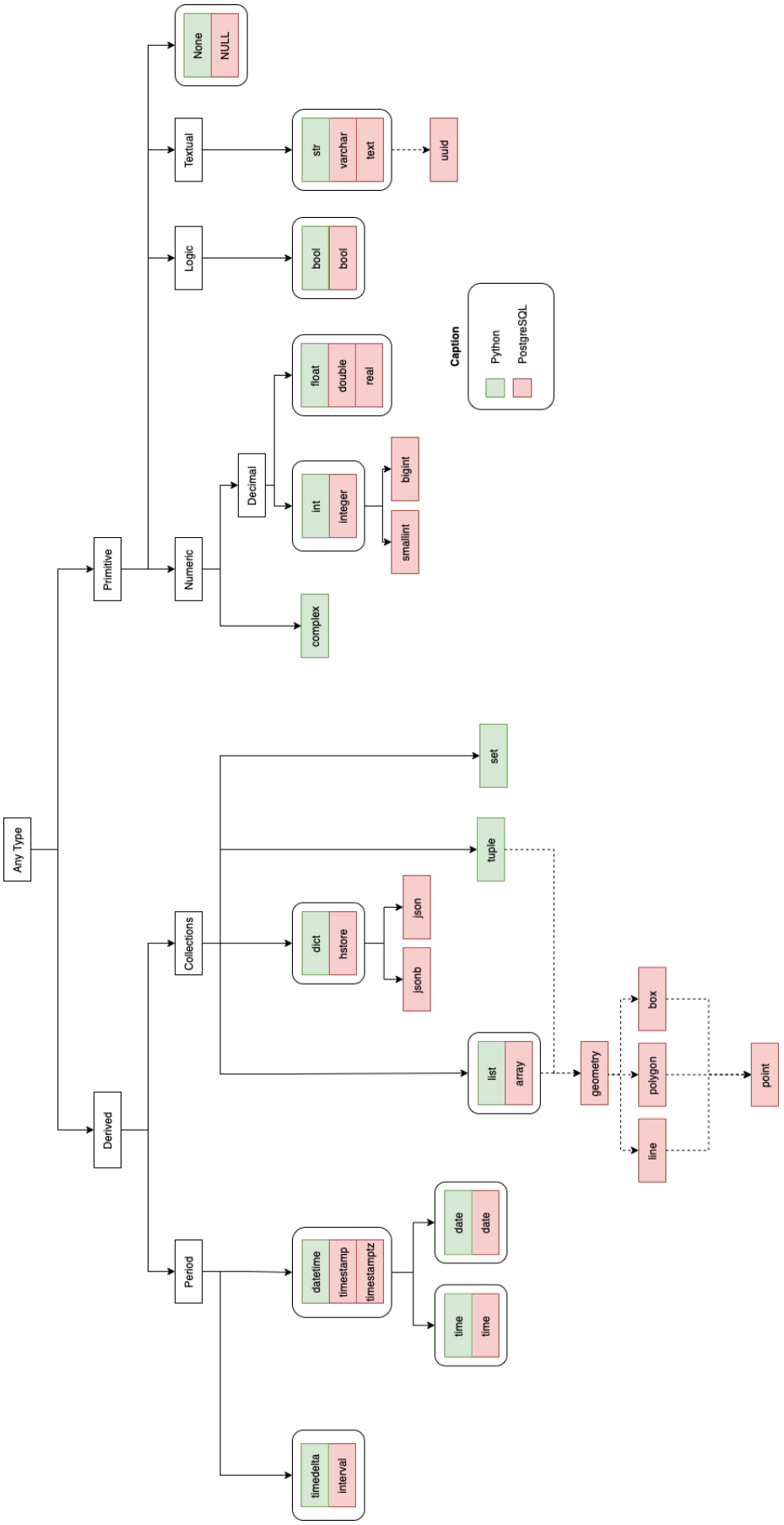

Figure 6.2: System's data type hierarchy

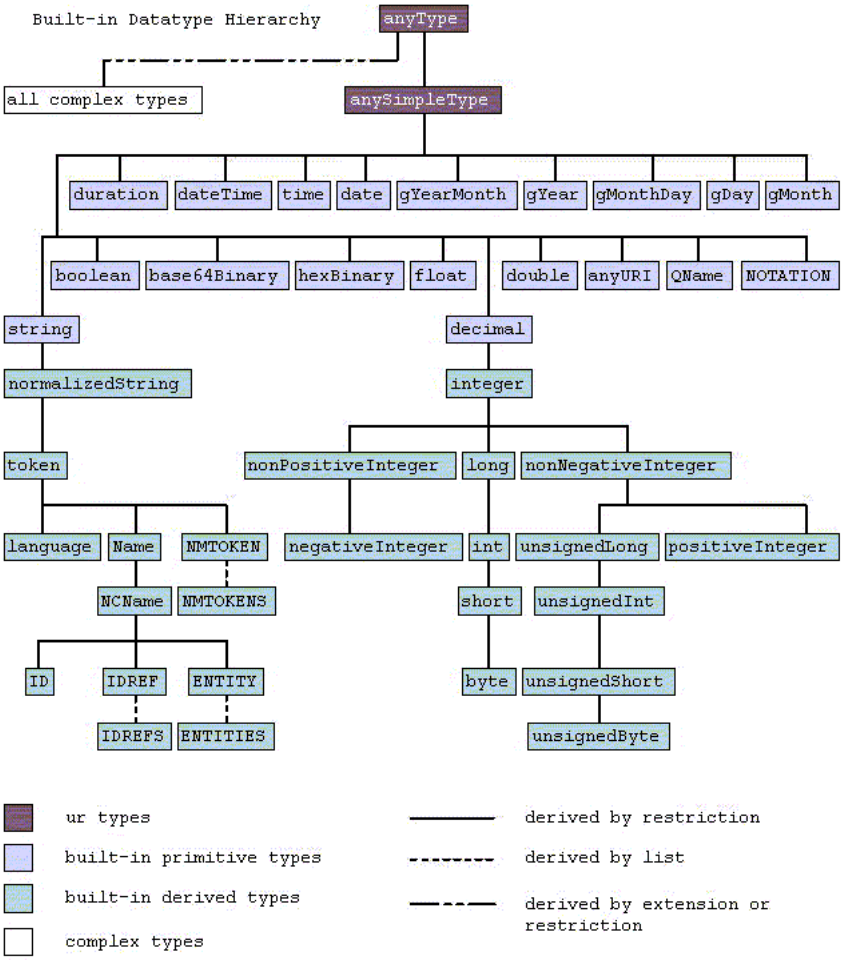

Figure 6.3: W3C's data type hierarchy [110]

Lastly, external sources of knowledge always help when there is not much information available. WordNet was the chosen thesaurus to support this similarity measure. For each leaf attribute it compares the WorNet synsets and, if the attribute has multiple terms and the whole expression does not have an entry on WordNet, it calculates the mean of the similarities between the terms that form the attributes. The synset comparison is done by firstly fetching both attributes' synsets. Then, the Wu and Palmer similarity is calculated for each pair of attributes and the highest similarity value is returned. The whole method is specified in Algorithm 17.

6.2.5 Mapping Module

Mapping Module deals with the matrices returned after the *Parallel Module* and delivers the final matches. The first step is the combination of all four matrices from the four parallel matchers. Since all matrices have the same dimensions and each position represents the same pair being compared, for each index an operation is performed, for instance, the mean, maximum or minimum. The library *numpy* offers a variety of these functions. The result is a final matrix which combines all four results in some way, as Algorithm 18 demonstrates.

To give the final verdict, the candidate selection performed earlier is taken into account and a weight (one or two) is multiplied to each value. In the end, a mapping is added to the final list of matches if the its score falls under a certain condition, for example, the

Algorithm 16 Type matcher

```
function typeMatch(hierarchy, alpha, beta, candidates, cand_matrix, shared_var)
    sim_matrix ← zerosMatrixShape(cand_matrix)
    for cand_list in candidates do          ▷ There are two candidate lists: one for good candidates and one for the best candidates
        for cand in cand_list do
            type1 ← cand[0][length(cand[0] − 1)][1]   ▷ the leaf attribute data type of the first path
            type1 ← cand[1][length(cand[1] − 1)][1]   ▷ the leaf attribute data type of the second path
            s ← typeSimilarity(hierarchy, alpha, beta, type1, type2)
            pos ← cand[2]                   ▷ checks the position of the pair on the matrix
            sim_matrix[pos] ← sim
        end for
    end for
    shared_var.insert(sim_matrix)
end function

function typeSimilarity(hierarchy, alpha, beta, type1, type2)
    if type1 != type2 then
        l ← shortestPathLength(hierarchy, type1, type2)
        lca ← lowestCommonAncestor(hierarchy, type1, type2)
        h ← shortestPathLength(hierarchy, "AnyType", lca)
        f ← exp(−beta ∗ l)
        g ← (exp(alpha ∗ h) − exp(−alpha ∗ h))/(exp(alpha ∗ h) + exp(−alpha ∗ h))
        sim ← f ∗ g
    else
        sim ← 1
    end if
    return sim
end function
```

Algorithm 17 Reuse matcher

```
function reuseMatch(candidates, cand_matrix, shared_var)
    sim_matrix ← zerosMatrixShape(cand_matrix)
    for cand_list in candidates do        ▷ There are two candidate lists: one for good candidates and one for the best candidates
        for cand in cand_list do
            w1 ← cand[0][length(cand[0] − 1)][0]    ▷ the leaf attribute name of the first path
            w2 ← cand[1][length(cand[1] − 1)][0]    ▷ A normalization may be applied if desired
            s ← compareSynsets(w1, w2)
            if s = 0 then
                elem_sims ← [ ]
                w1 ← split(w1, ' ')
                w2 ← split(w2, ' ')
                for elem1 in w1 do
                    for elem2 in w2 do d ← compareSynsets(elem1, elem2)
elem_sims.insert(d)
                    end for
                end for
                s ← mean(elem_sims)
            end if
            pos ← cand[2]                ▷ checks the position of the pair on the matrix
            sim_matrix[pos] ← s
        end for
    end for
    shared_var.insert(sim_matrix)
end function


function compareSynsets(word1, word2)
    synsets1 ← getSynsets(word1)
    synsets2 ← getSynsets(word2)
    scores ← [ ]
    for syn1 in synsets1 do
        for syn2 in synsets2 do
            s ← wuPalmerSim(syn1, syn2)
            scores.insert(s)
        end for
    end for
    if scores then
        sim ← max(scores)
    else
        sim ← 0
    end if
    return sim
end function
```

Algorithm 18 Combination and verdict

function COMBINATIONVERDICT($matrices, candidates, threshold$)
  $axis \leftarrow 0$
  $final\_matrix \leftarrow mean(matrices, axis)$
  $cand_num \leftarrow 0$
  $final\_mappings \leftarrow [\,]$
  for $cand\_list$ in $candidates$ **do** ▷ There are two candidate lists: one for *good* candidates and one for the *best* candidates
    $cand_num \leftarrow cand_num + 1$
    for $cand$ in $cand\_list$ **do**
      $pos \leftarrow cand[2]$ ▷ checks the position of the pair in the matrix
      $final\_matrix[pos] \leftarrow final\_matrix[pos] * cand\_num$
      if $final\_matrix[pos] > threshold$ **then**
        $final\_mappings.insert(cand)$
      end if
    end for
  end for
  return $final\_matrix$, $final\_mappings$
end function

similarity being above a threshold.

It is important to note that the system is able to output $1:1$ and $n:1$ mappings, achieving a global cardinality of $?:*$.

6.2.6 Statistics Module

The auxiliary *Statistics Module* provides backup functions, making it possible to visualize and analyze the whole intermediate stages of the pipeline. The first function shows the graph. The Python code to replicate the graph observable in Figure 6.4 can be found in Listing 6.8.

```
import pydot
from networkx.drawing.nx_pydot import graphviz_layout
import matplotlib.pyplot as plt

#function to plot a graph
def plotGraph(G):

    #shows graph in a top-down tree format
    pos = graphviz_layout(G, prog= "dot")
    nx.draw_networkx(G, pos, with_labels=True)
    plt.show()

    return
```

Listing 6.8: Python code to plot a graph

The second algorithm (Algorithm 19) returns the candidate pairs selected. The rows and columns of the candidate matrix represent the attribute paths in the file and in the database, respectively, that are being compared. If the element is equal to one, it means that the pair is a good matching candidate. In addition, if the value is equal to two, it means that the pair was determined a high probability candidate. These two lists of candidates are the output of this function.

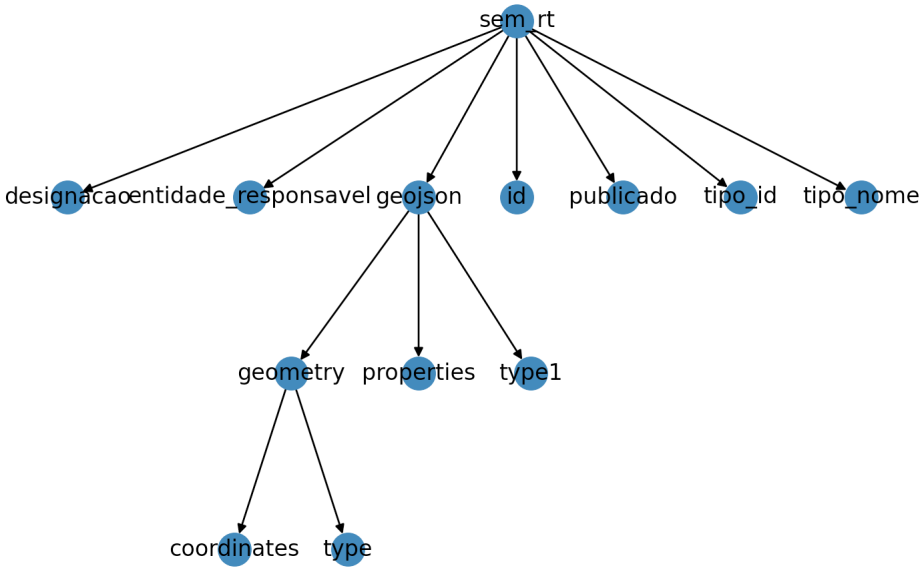

Figure 6.4: Graph example

The third function plots the heat map, showing similarity matrices in a more pleasant way. Once again, the Python code to replicate the image observable in Figure 6.5 can be found in Listing 6.9.

```
import seaborn as sns
import matplotlib.pylab as plt

#function to show a similarity matrix in a more pleasant way
def heatMap(matrix, labely, labelx):
    plt.style.use("seaborn")

    #Plots the heatmap
    plt.figure(figsize=(10,10))
    heat_map = sns.heatmap(matrix,  xticklabels=labelx, yticklabels=
    labely, linewidth=1 , annot=True)
    plt.title("Similarity matrix")
    plt.show()

    return
```

Listing 6.9: Python code to plot a heat map

The last function (Algorithm 20) calculates all the necessary scores to evaluate the performance of the matcher. For that, it consults the mapping files, similar to Listing 6.10, created for testing. These are JSON files with the file attribute as key and the correspondent database columns as value. It crawls over all the mappings and counts the true positives. Next, it calculates the other values, and finally the scores.

```
{
    "ExternalIdentification": ["offstreet_spots_offstreetparkingarea","
    external_id"],
    "Designation": ["offstreet_spots_offstreetparkingarea", "name"],
    "description": ["offstreet_spots_offstreetparkingarea", "description
    "],
    "street": ["offstreet_spots_offstreetparkingarea", "address"]
}
```

Listing 6.10: Sample of a mapping file

Algorithm 19 Candidate information

function CANDIDATEINFO($file\_paths, db\_paths, cand\_matrix$)
  $good \leftarrow [\,]$
  $best \leftarrow [\,]$
  for $row \leftarrow 0$ **to** $length(file\_paths) - 1$ **do**
    for $col \leftarrow 0$ **to** $length(db\_paths) - 1$ **do**
      if $cand\_matrix[row][col] = 2$ **then**
        $best.insert([file\_paths[row], db\_paths[col], [row, col]])$ ▷ The candidate position in the matrix is also added to help future matrix queries
      else if $cand\_matrix[row][col] = 1$ **then**
        $good.insert([file\_paths[row], db\_paths[col], [row, col]])$
      end if
    end for
  end for
  return $good, best$
end function

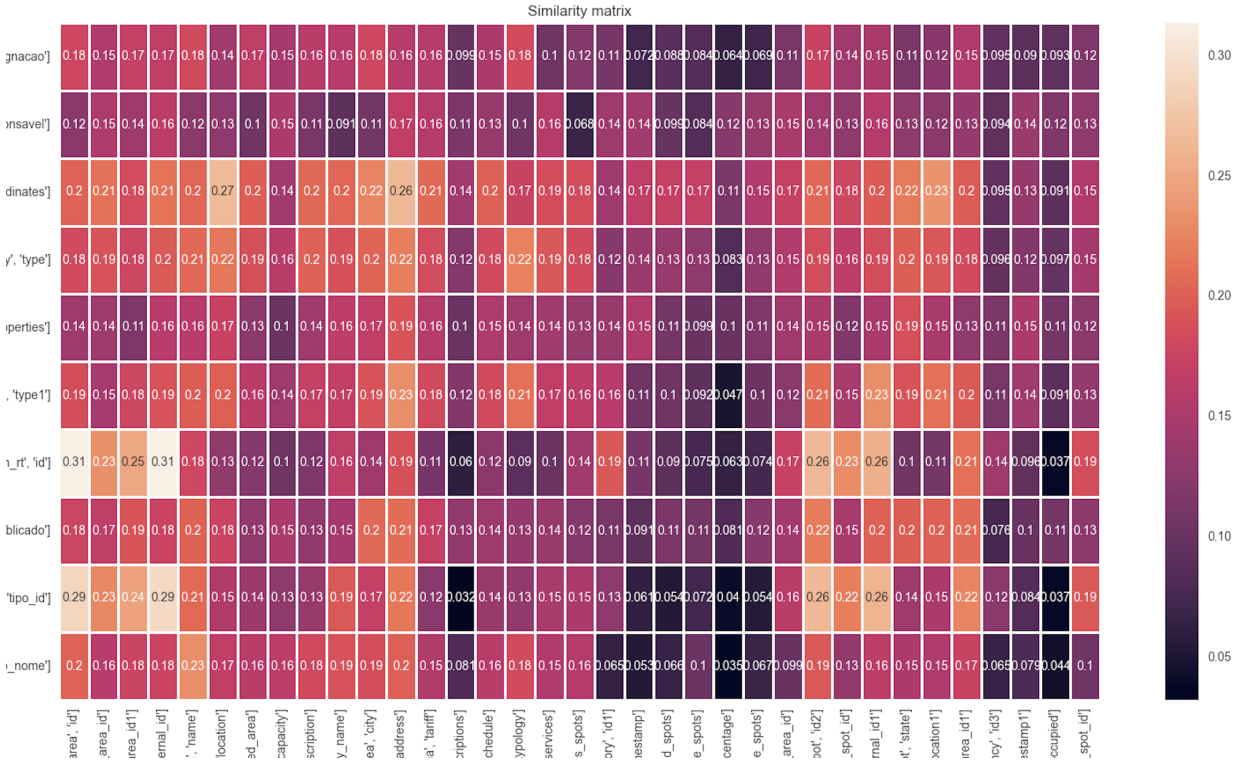

Figure 6.5: Heat map example

6.3 Conclusion

It was shown in this chapter how to reproduce a similar matching system. The explanation was grouped by modules and the algorithms were presented in pseudo-code to be transversal to all coding languages and frameworks.

The next chapter demonstrates how were the tests conducted, as well as interpreting their results.

Algorithm 20 Scores calculation

```
function getScores(final_mappings, mapping_name, file_paths, db_paths)
    mapping_file ← importFile(mapping_name)
    tp ← 0
    num_mappings ← 0
    num_final_mappings ← 0
    for key in mapping_file do
        num_mappings ← num_mappings + 1
        for map in final_mappings do
            num_final_mappings ← num_final_mappings + 1
            if map[0] = key and map[1] = mapping_file[key] then
                tp ← tp + 1
            end if
        end for
    end for
    total ← length() * lentgh()
    fp ← num_final_mappings − tp
    fn ← num_mappings − tp
    tn ← total − tp − fp − fn
    if tp + fp > 0 then
        precision ← tp/(tp + fp)
    else
        precision ← 0
    end if
    if fn + tp > 0 then
        recall ← tp/(fn + tp)
    else
        recall ← 0
    end if
    if precision + recall > 0 then
        f1_score ← (2 * (precision * recall))/(precision + recall)
    else
        f1_score ← 0
    end if
    return precision, recall, f1_score
end function
```

Chapter 7

Experimental Study

Experimentation was an essential part of this project. During and after the development tests were performed to ensure that every component worked properly and to maximize results within known options that could potentially bring better results. This empirical study is equally important to evaluate if the solution developed meets the performance requirements previously established and to provide important clues for future improvements.

That being said, this chapter presents how the dataset was put together, the experiment setup and procedure followed to compare different configurations, and the results obtained.

7.1 Dataset

Assembling the dataset to validate the solution was a big challenge. For the most part, because Ubiwhere did not have that many sources yet to compose a robust dataset. The lack of open-source data which fitted UBP's data models, was also an obstacle. For those reasons, it was decided that it would be reasonable to create a dataset with real world files mixed with fictional files, generated according to what was expected from real sources.

To simplify and standardize the tests, it was decided that the files would be loaded off a local folder instead of requesting some of them through sensor APIs and loading the artificial ones. This makes the dataset more viable in the long term, since it is not dependent on any API maintenance.

Having all that in mind, the testing dataset, having a total of twenty files was split in two parts. The first half, named *Easy Cases*, focus on simpler cases of the UBP, in other words, simpler schemas, with less attributes with more or less direct mappings to a single database table (Figure 7.1). The second half, on the other hand, is called *Hard Cases*, because it aggregates larger and harder examples, whose mappings are not always trivial (Figure 7.2). To add an extra factor of realism to the artificial sources, attributes in languages such as Portuguese, French and Spanish were mixed with the English ones

For each part of the dataset, the database tables that would take part of the test were chosen for how easy would be the isolation of the problem from other challenges inherent to the UBP's data model conversion, such as the matching of tables, but also for how easy would it be to find or build examples that fitted those same tables. Some tables had already very straightforward mappings which did not justify to be part of the testing dataset. Others had columns whose values were the result of operations with other tables' columns, making the case unnecessarily complex and diverting focus from the problem that

Figure 7.1: *Easy Cases:* example file

was being studied.

For *Easy Cases*, the database tables chosen to integrate the experiment, were the air quality reading and the radar reading tables (Figure 7.3). For *Hard Cases* the tables selected were all tables related with parking (Figure 7.4), parking events and occupancy history. Most difficult cases arose from the perspective of evaluating the extent to which it was possible to map to different tables even within the same context, parking related data in this case.

7.2 Experimental Setup

The purpose of the testing phase was to properly tune the system, as well as evaluating it and getting justified clues on what and how it can be improved. *Statistics Module* played an important role, as it provided the functions needed to calculate the scores and to perform the analysis. A new branch was also opened on the repository, so that eventual necessary changes to the code could be done without losing the normal system setting.

The experiment took place entirely on the development computer, a device equipped with a 2.5 GHz *Intel Core i7* dual core, 16 GB of RAM and an *Intel Iris Plus Graphics 640*. The internet connection supplied via Ethernet had 200Mbps and 120Mbps download and upload speeds respectively[1].

The tests themselves were focused on the following aspects:

Threshold: The threshold is the edge separating a match from a non-matching pair. After computing the final matching it is necessary to give a verdict to each pair. For this system a static threshold was set manually. The key to a good threshold is a value that balances well the true positives and the false positives.

N-grams: As explained in the above chapter the *Parallel Matching Module* has a token-based matcher, which splits attributes into smaller chunks and then compares them. These sub-strings can have different lengths that can affect the final similarity calculated.

[1]measured with: https://www.speedtest.net/

```
"ParkingList":{
  "Parking":[
    {
      "ParkingCode":2026,
      "Name":"Ona Glòries",
      "Address":"C/ de la Ciutat de Granada, 173-175",
      "ParkingAccess":{
        "Access":{
          "AccessID":14,
          "AccessAddress":"C/ de la Ciutat de Granada, 171",
          "Longitude":2.18961,
          "Latitude":41.404657
        }
      },
      "MaxWidth":"None",
      "MaxHeight":2.2,
      "Guarded":1,
      "InformationPoint":1,
      "Open":"00:00",
      "Close":"24:00",
      "Exterior":0,
      "HandicapAccess":1,
      "ElectricCharger":1,
      "WC":1,
      "Elevator":1,
      "Consigna":1,
      "ParkingPriceList":{
        "Price":[
          {
            "VehicleType":"Moto",
            "FractionType":"Normal",
            "DescFractionType":"0001",
            "Minutes":1,
            "Amount":0.0068
          },
          {
            "VehicleType":"Turisme",
            "FractionType":"Normal",
            "DescFractionType":"0001",
            "Minutes":1,
            "Amount":0.054644
          }
        ]
      },
      "ReferenceRate":16.0,
      "Ownership":1,
      "ParkingType":"01"
    },
```


Figure 7.2: *Hard Cases:* example file

String Similarity: String similarity also takes place during the parallel matching phase. There are many measures, but the two most widely adopted are the Levenshtein and Jaro-Winkler's.

Normalization: Throughout the course of this project several normalization functions were implemented in an attempt to tackle some of the obstacles raw attributes had. These functions include a simple normalization function, a normalization with language detection and translation, and a function with lemmatization added on top of the other normalization functions.

It is important to mention that the combination method was kept the same throughout the whole experiment, since evidences have been found in the literature showing the mean as the best option for most cases, as it captures equally the best and the worse of all matchers [113]. The constants α and β were also kept 0.3057 according to the recommendations of the method's original paper [78]. Therefore, the study comprises three smaller experiments, one for tuning the threshold, other for tuning the parallel matchers' parameters

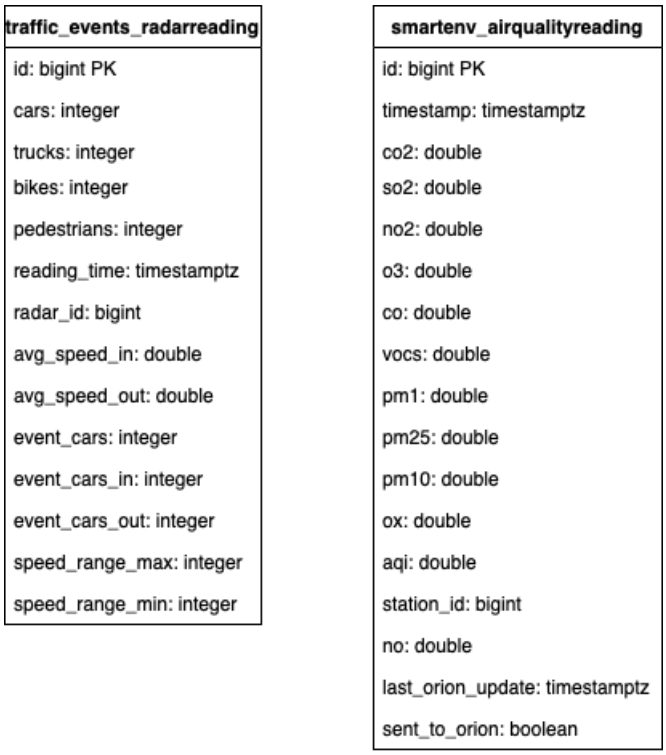

Figure 7.3: *Easy Cases:* database tables

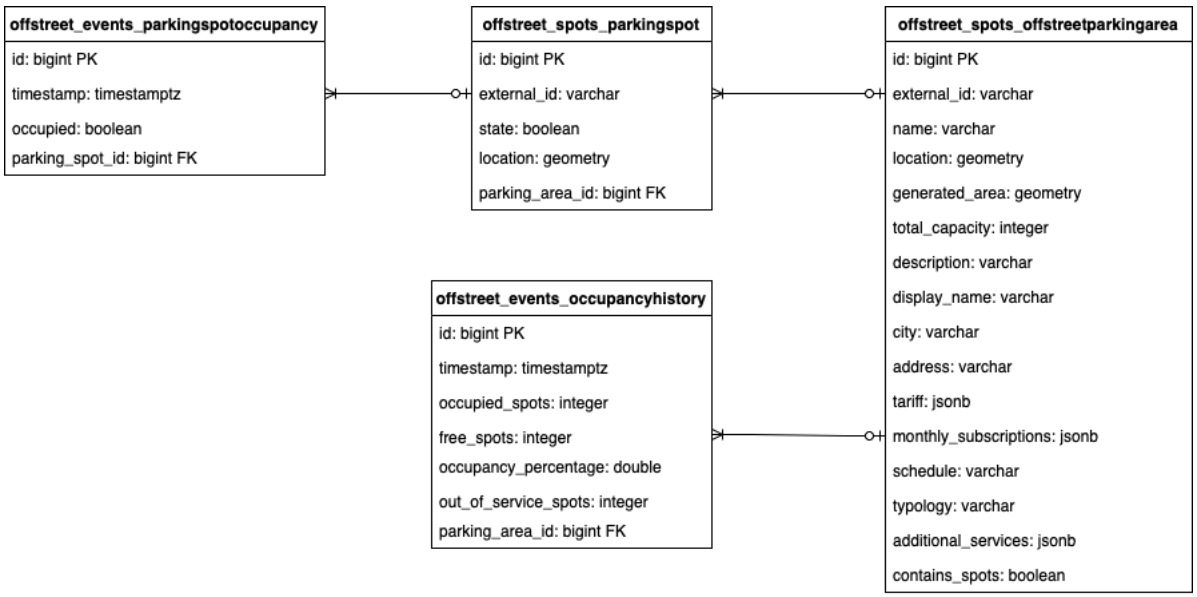

Figure 7.4: *Hard Cases:* database tables

and the last one for evaluating the normalization functions, which will be detailed in the upcoming sections.

7.3 Threshold

The first set of tests had the objective of finding the best threshold possible. It is hard to separate and evaluate each parameter individually in a non biased way, because the

final result is shaped by all of them. To overcome this obstacle, three distinct cases were defined, where the number of grams (two, three and four), string similarity measure (Levenshtein and Jaro-Winkler) and normalization method (without normalization, simple, with translation and with translation and lemmatization) were randomly combined and tested for each threshold. Thereby, the scores could be compared and it became possible to have a clear idea of how different thresholds influence the system's performance. The base settings picked for testing the thresholds were:

Two grams; Levenshtein similarity; no normalization.

Three grams; Levenshtein similarity; normalization with translation.

Three grams; Jaro-Winkler's similarity; simple normalization.

The first thresholds being tested were 0.25, 0.50 and 0.75, so to bracket the window within what the ideal threshold would be. A preliminary analysis points out to the conclusion that smaller thresholds work better than higher ones, since the combination of every matrix, each one of them focusing in certain aspects of the attribute, produces relatively small values for the majority of the ambiguous cases (Figure 7.5).

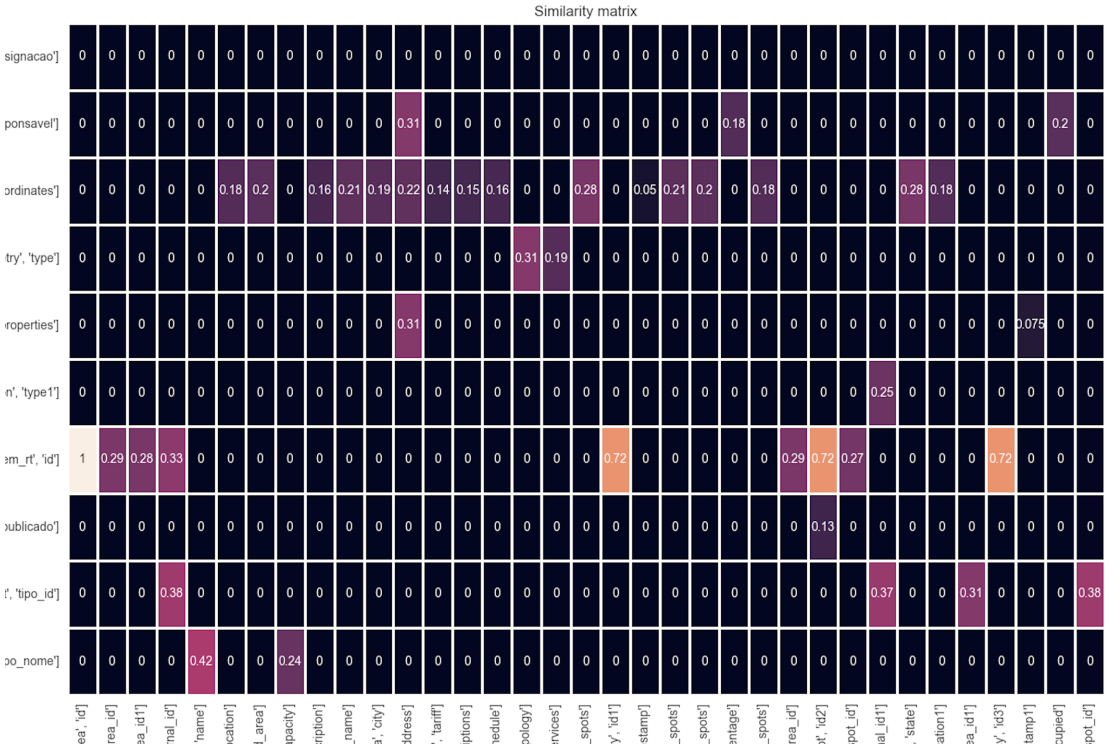

Figure 7.5: Example of a matrix resulted from all combined matrices

The plots below show the scores obtained for each threshold (Figures 7.6, 7.7 and 7.8). On the yellow bar is displayed the precision, the blue shows the recall, the green the f1-score and the red exhibits the percentage of the testing attributes that match a column of a table in the database. This last percentage is just another characteristic which dictates the difficulty of the matching schemas, evidenced by the *Hard Cases*, which have a smaller percentage of attributes with an actual match.

Each plot expresses the mean of the files' scores for each threshold, either for the *Easy Cases* (Figure 7.6) and the *Hard Cases* (Figure 7.7) separately (ten files for each category), or the mean of all twenty cases, referred as "overall" (Figure 7.8).

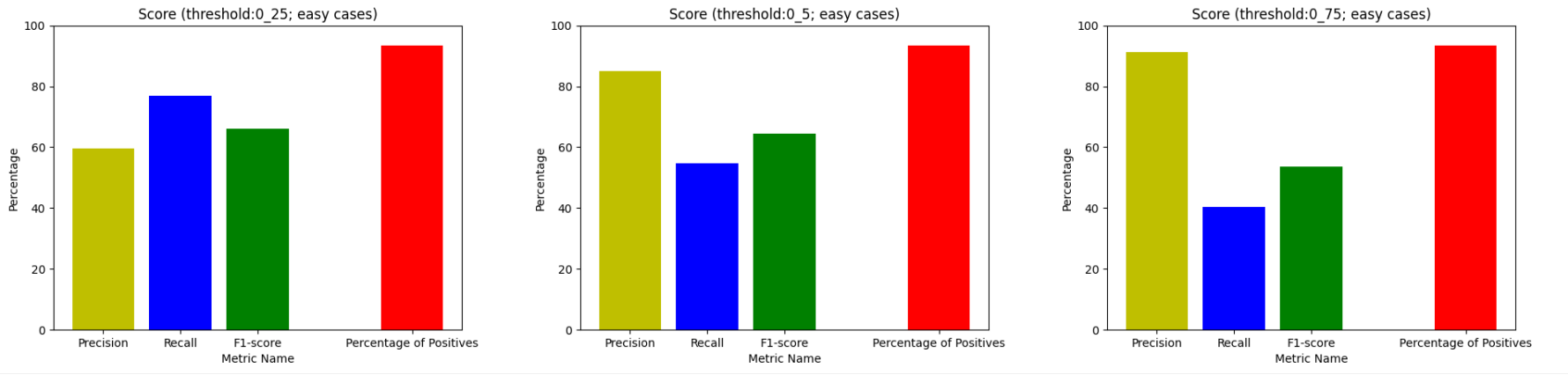

Figure 7.6: Scores for the thresholds 0.25, 0.50 and 0.75 (*Easy Cases*)

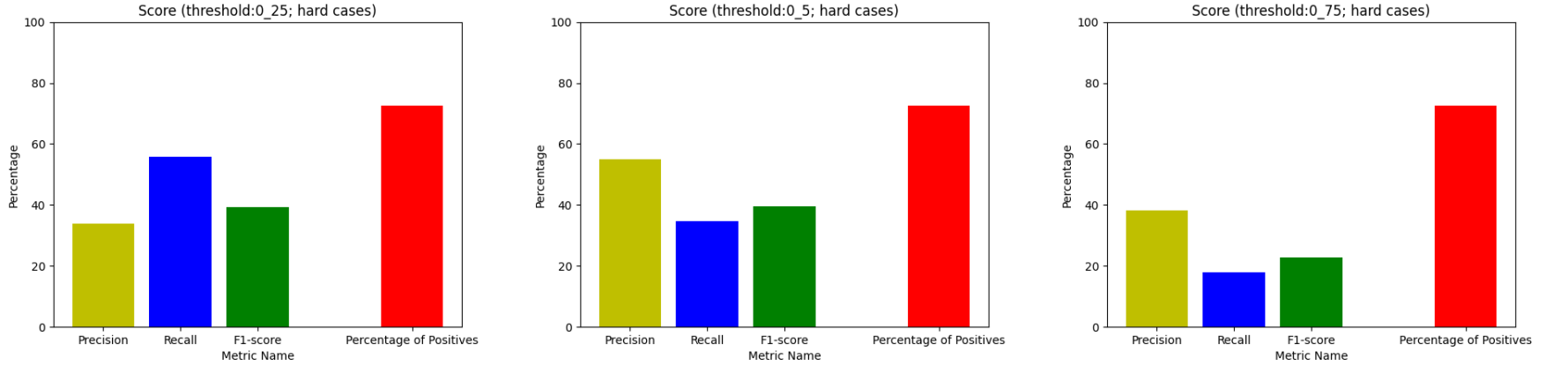

Figure 7.7: Scores for the thresholds 0.25, 0.50 and 0.75 (*Hard Cases*)

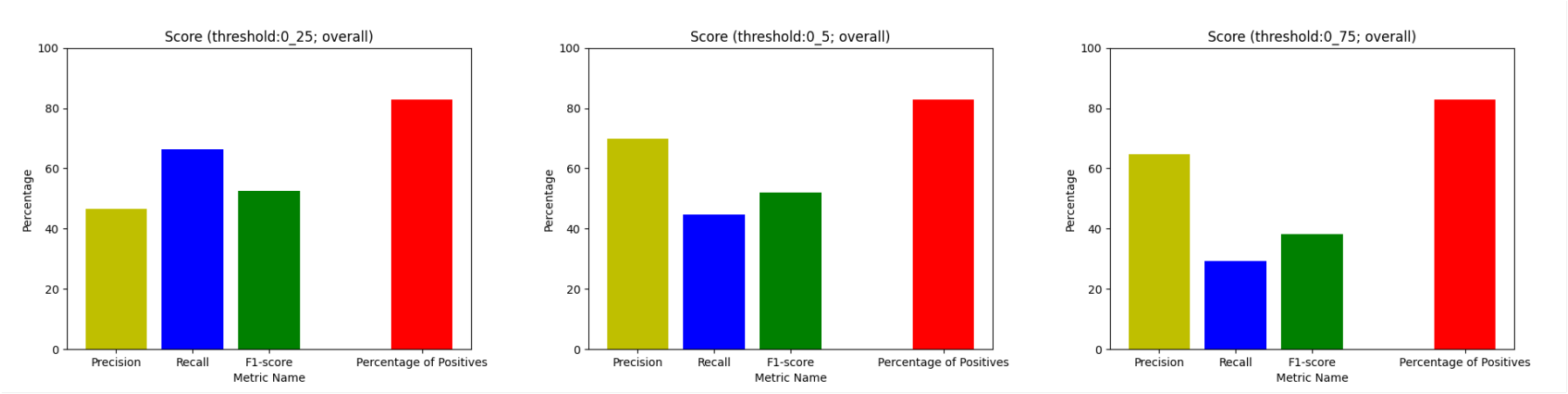

Figure 7.8: Scores for the thresholds 0.25, 0.50 and 0.75 (overall)

Threshold	Mean Precision (%)	Mean Recall (%)	Mean F1-score (%)	Mean Runtime (min.)
Easy Cases (93.37% of matching attributes in average)				
0.25	59.55	76.85	66.02	0.29
0.50	84.88	54.58	64.57	0.32
0.75	91.28	40.49	53.66	0.17
Hard Cases (72.48% of matching attributes in average)				
0.25	33.81	55.85	39.2	1.83
0.50	55.06	34.57	39.53	1.28
0.75	38.11	17.85	22.83	0.76
Overall (82.93% of matching attributes in average)				
0.25	46.68	66.35	52.61	1.06
0.50	69.97	44.57	52.05	0.80
0.75	64.69	29.17	38.24	0.47

Table 7.1: Sum up of the first threshold experiment

The results, summarized in Table 7.1, confirm what had been observed in the produced matrices during the *Mapping Module*. The best scores are reached when choosing thresholds bellow 0.50. Scores for the *Easy Cases* are better in average than for the *Hard Cases*, as expected, but they both react in the same way to the different thresholds, showing 0.25 and 0.50 the best performances.

Although the focus of this test suite has been the scores, the average running times were controlled to make sure the execution stayed within the temporal limits imposed by the non-functional requirements, as it can be seen in Table 7.1.

Due to the similarity of the f1-score values for thresholds 0.25 and 0.50, a second test suite was done to more precisely determine the best threshold. This time, the chosen values were 0.313, 0.375 and 0.438, being, therefore, within 0.25 and 0.50.

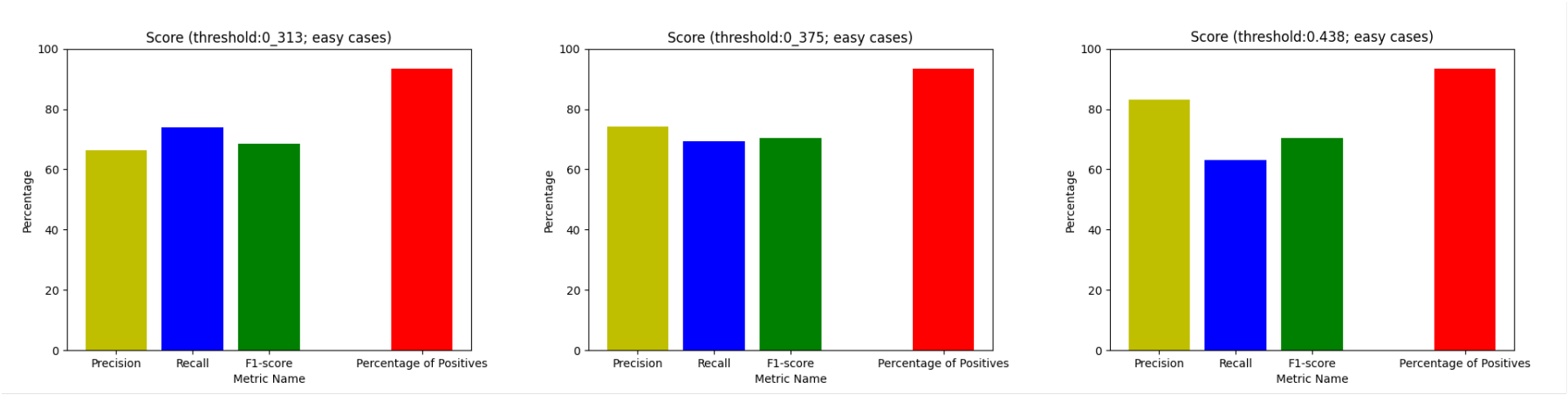

Figure 7.9: Scores for the thresholds 0.313, 0.375 and 0.438 (*Easy Cases*)

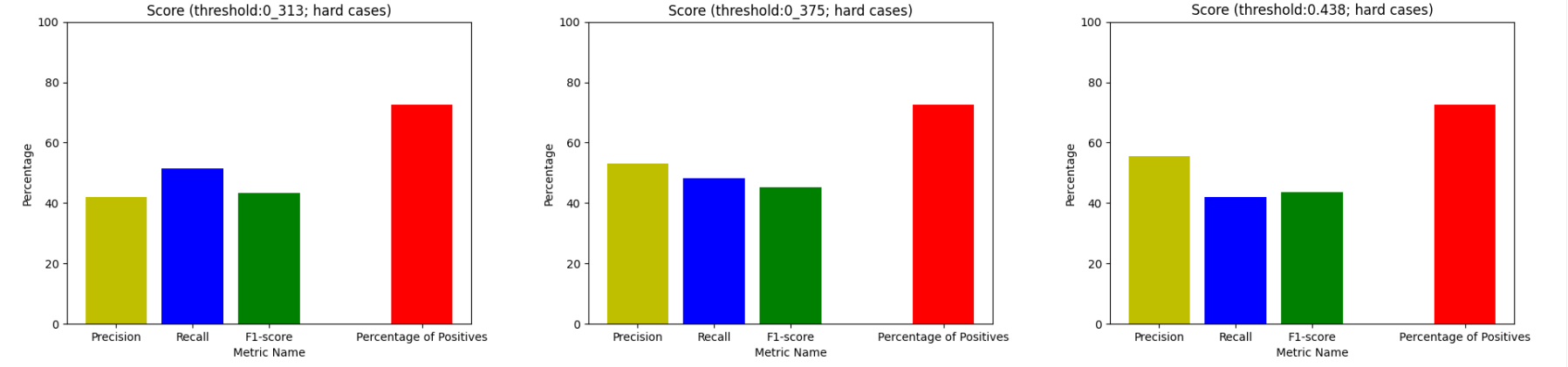

Figure 7.10: Scores for the thresholds 0.313, 0.375 and 0.438 (*Hard Cases*)

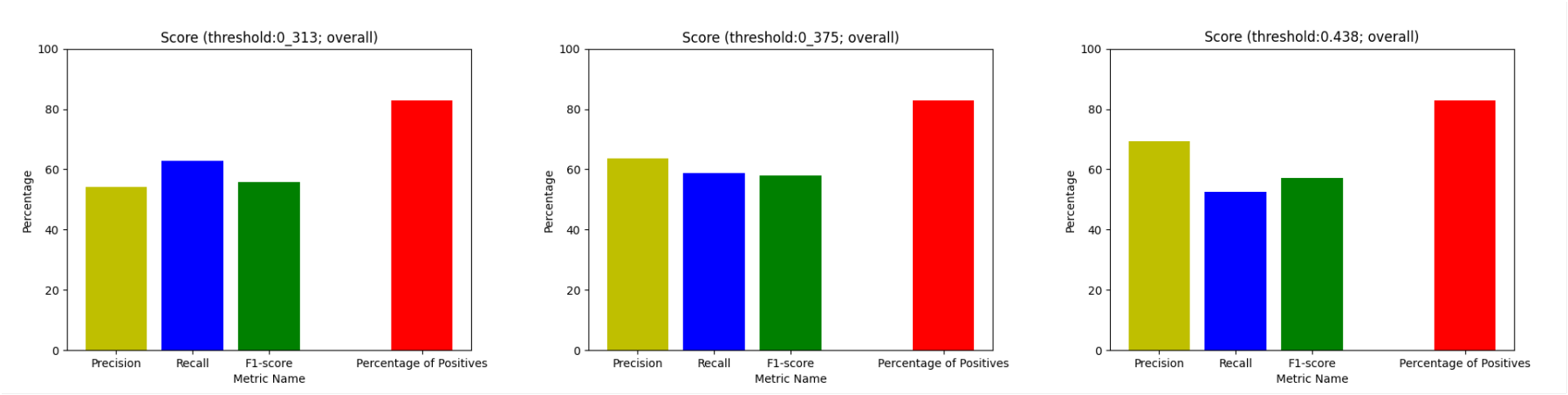

Figure 7.11: Scores for the thresholds 0.313, 0.375 and 0.438 (overall)

Threshold	Mean Precision (%)	Mean Recall (%)	Mean F1-score (%)	Mean Runtime (min.)
Easy Cases (93.37% of matching attributes in average)				
0.313	66.31	73.94	68.48	0.15
0.375	74.27	69.2	70.39	0.16
0.438	83.15	63.14	70.46	0.15
Hard Cases (72.48% of matching attributes in average)				
0.313	42.02	51.52	43.23	0.38
0.375	53.11	48.12	45.3	0.39
0.438	55.6	41.89	43.56	0.38
Overall (82.93% of matching attributes in average)				
0.313	54.16	62.73	55.86	0.26
0.375	63.69	58.66	57.85	0.27
0.438	69.37	52.52	57.01	0.27

Table 7.2: Sum up of the second threshold experiment

The results (see Figure 7.11 and Table 7.2) indicate that 0.375 is the best of the tested threshold values, as it corresponds to the best precision-recall compromise overall. That being said the chosen threshold was 0.375. This value was maintained as a reference for the remaining experimental study.

7.4 Tuning Parallel Matchers

Once fixed the threshold, the next phase of the experiment could begin, which involved testing the matcher's configurations during the parallel stage of the pipeline.

Thanks to its modularity, the code was easily adapted, so that the token and the string matchers, would not run simultaneously. Preventing these two matches to execute at the same time, isolated the two matchers, allowing the different parameters to be tested individually without interfering with one another. Of course that by not executing all matchers, the scores become inevitably lower, but the goal here was not the absolute percentages, but rather compare scores among configurations, to check which one performed better.

Starting by the number of grams in the token matcher, three values were tried initially, being them two, three and four. For each threshold, four tests were made, in order to cover all normalization options. Once again, the results are presented in the form of bar charts. They show the average scores for *Easy Cases* (Figures 7.12), *Hard Cases* (Figures 7.13) and the overall mean for each tested number (Figures 7.14).

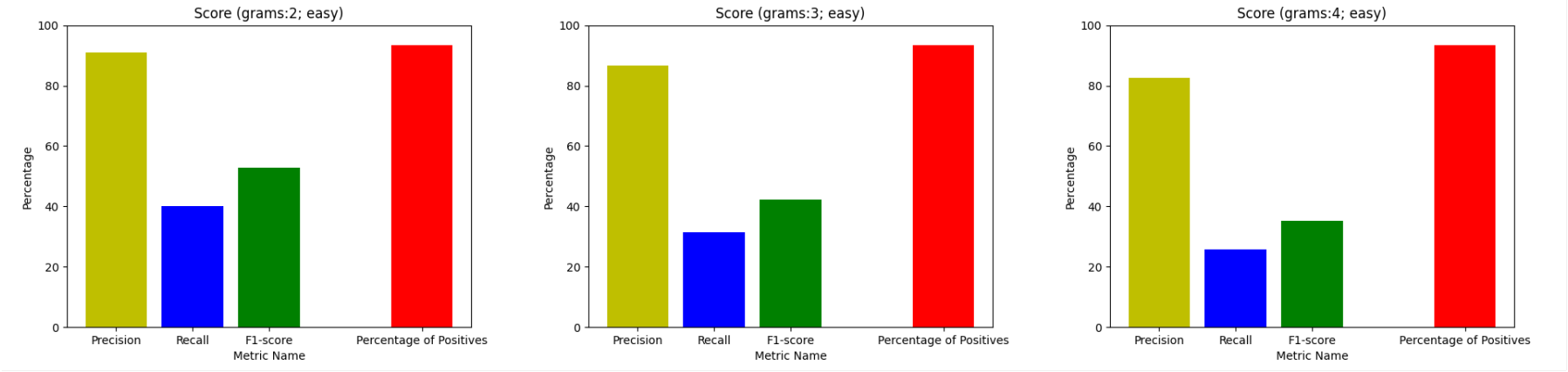

Figure 7.12: Scores for the grams two, three and four (*Easy Cases*)

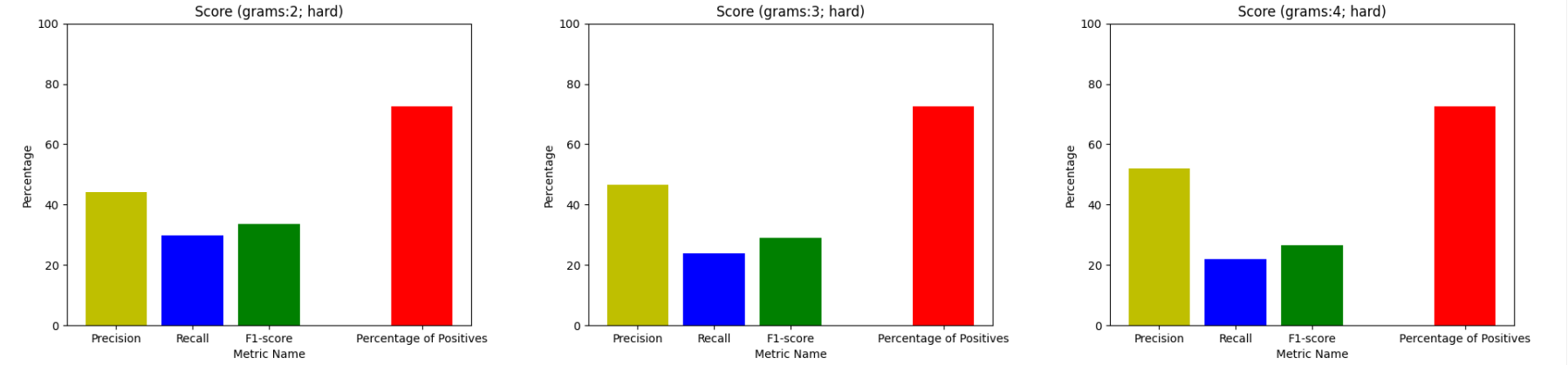

Figure 7.13: Scores for the grams two, three and four (*Hard Cases*)

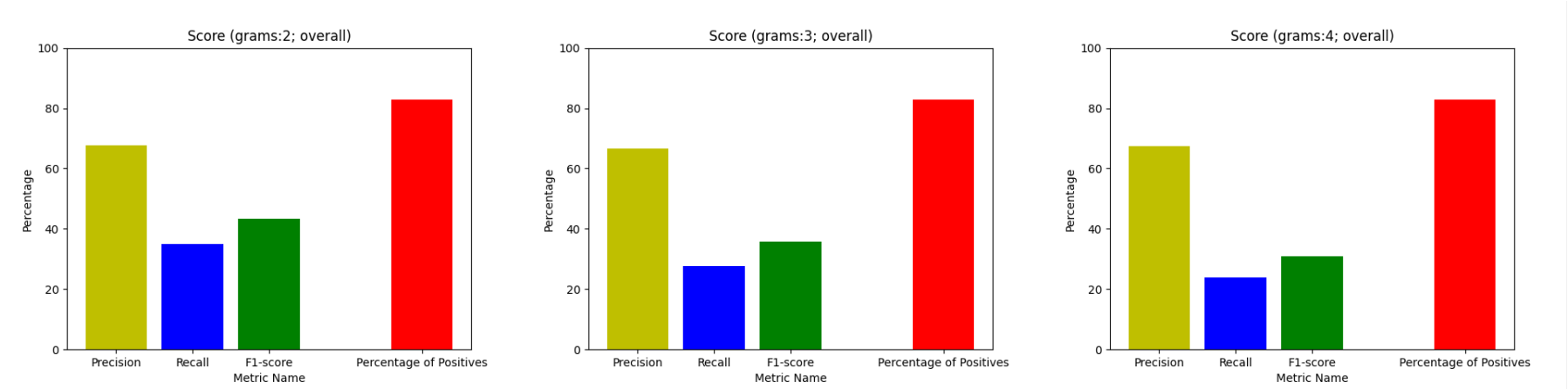

Figure 7.14: Scores for the grams two, three and four (overall)

N	Mean Precision (%)	Mean Recall (%)	Mean F1-score (%)	Mean Runtime (min.)
Easy Cases (93.37% of matching attributes in average)				
2	91.04	40.01	52.91	0.12
3	86.67	31.4	42.3	0.12
4	82.5	25.65	35.3	0.12
Hard Cases (72.48% of matching attributes in average)				
2	44.27	29.86	33.7	0.35
3	46.49	23.9	29.02	0.36
4	52.08	22.04	26.47	0.35
Overall (82.93% of matching attributes in average)				
2	67.66	34.94	43.31	0.23
3	66.58	27.65	35.66	0.24
4	67.29	23.84	30.88	0.23

Table 7.3: Sum up of the n-gram experiment

The results, summed up in Table 7.3, show similar behaviours, for the same configuration, between *Easy Cases* and *Hard Cases*, although the differences in scores for different configurations are more pronounced in the *Easy Cases*. The best scores were accomplished with grams of length two. A possible explanation for it is the fact that bigger grams, like four-grams, force words to be way more similar, especially when dealing with small words, which makes the token similarity technique handle poorly spelling mistakes and variations in orthography. On the other hand, acronyms and abbreviations work better with short tokens like bigrams, because they are obviously more compact and most words' radicals are small too. Since four-grams showed the worst results and the runtimes remained under the limit, the ideal number of grams was set to two and no more tests were necessary.

The methodology used to test the string matcher was analogous to the one used for tuning the token matcher, so the token matcher was not executed during the string matcher's set of tests. The statistics calculated, seen in Figures 7.15, 7.16 and 7.17, demonstrate that Levenshtein measure is preferable over Jaro-Winkler's.

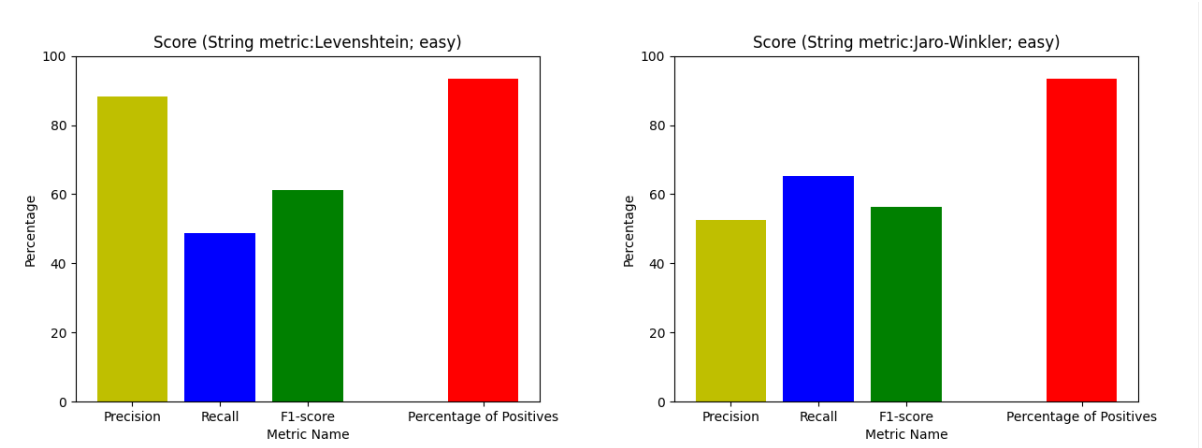

Figure 7.15: Scores for the Levenshtein and Jaro-Winkler's similarity measures (*Easy Cases*)

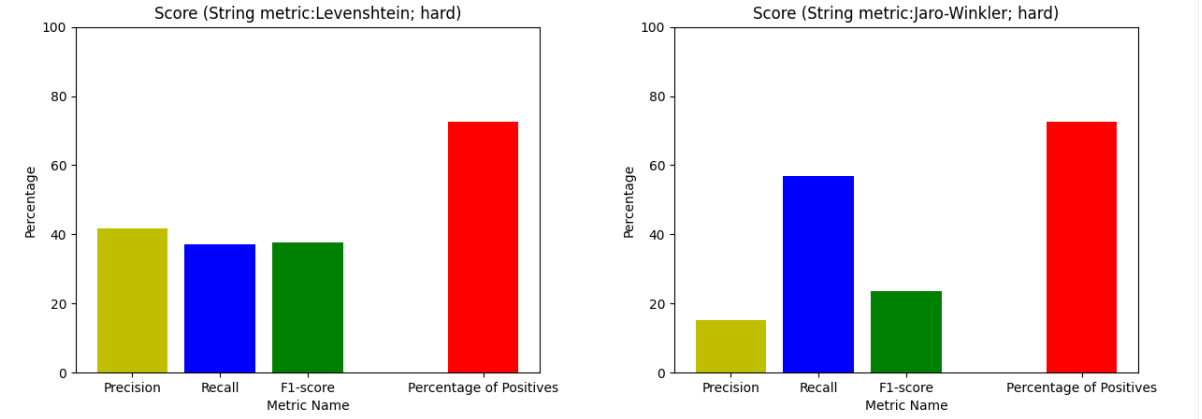

Figure 7.16: Scores for the Levenshtein and Jaro-Winkler's similarity measures (*Hard Cases*)

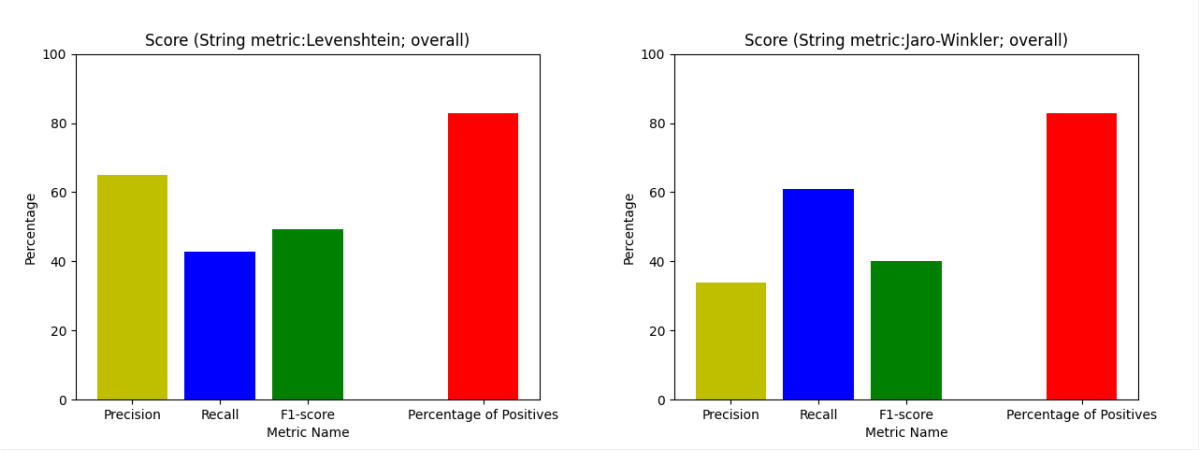

Figure 7.17: Scores for the Levenshtein and Jaro-Winkler's similarity measures (overall)

Measure	**Mean Precision (%)**	**Mean Recall (%)**	**Mean F1-score (%)**	**Mean Runtime (min.)**
Easy Cases (93.37% of matching attributes in average)				
Levenshtein	88.29	48.74	61.24	0.12
Jaro-Winkler	52.63	65.16	56.38	0.86
Hard Cases (72.48% of matching attributes in average)				
Levenshtein	41.6	37.1	37.56	0.35
Jaro-Winkler	15.14	56.82	23.68	0.35
Overall (82.93% of matching attributes in average)				
Levenshtein	64.95	42.92	49.4	0.24
Jaro-Winkler	33.88	60.99	40.03	0.61

Table 7.4: Sum up of the string similarity experiment

Although the best string similarity measure is difficult to assess and very dependent on the context in which it is applied, studies suggest the Jaro-Winkler similarity is better for short strings [49], such as first names, which is not the case for most attributes (e.g. attributes composed of two or more words: *parking_spot*). The average runtimes stood under the limit defined by the requirements, so there was no problem in that regard (Table 7.4).

7.5 Normalization

The last parameter to be tuned was the kind of normalization utilized. For this, the variables tested in the last sections were set to the values which provided the best scores, i.e. threshold 0.375, bigrams and Levenshtein as the string similarity measure.

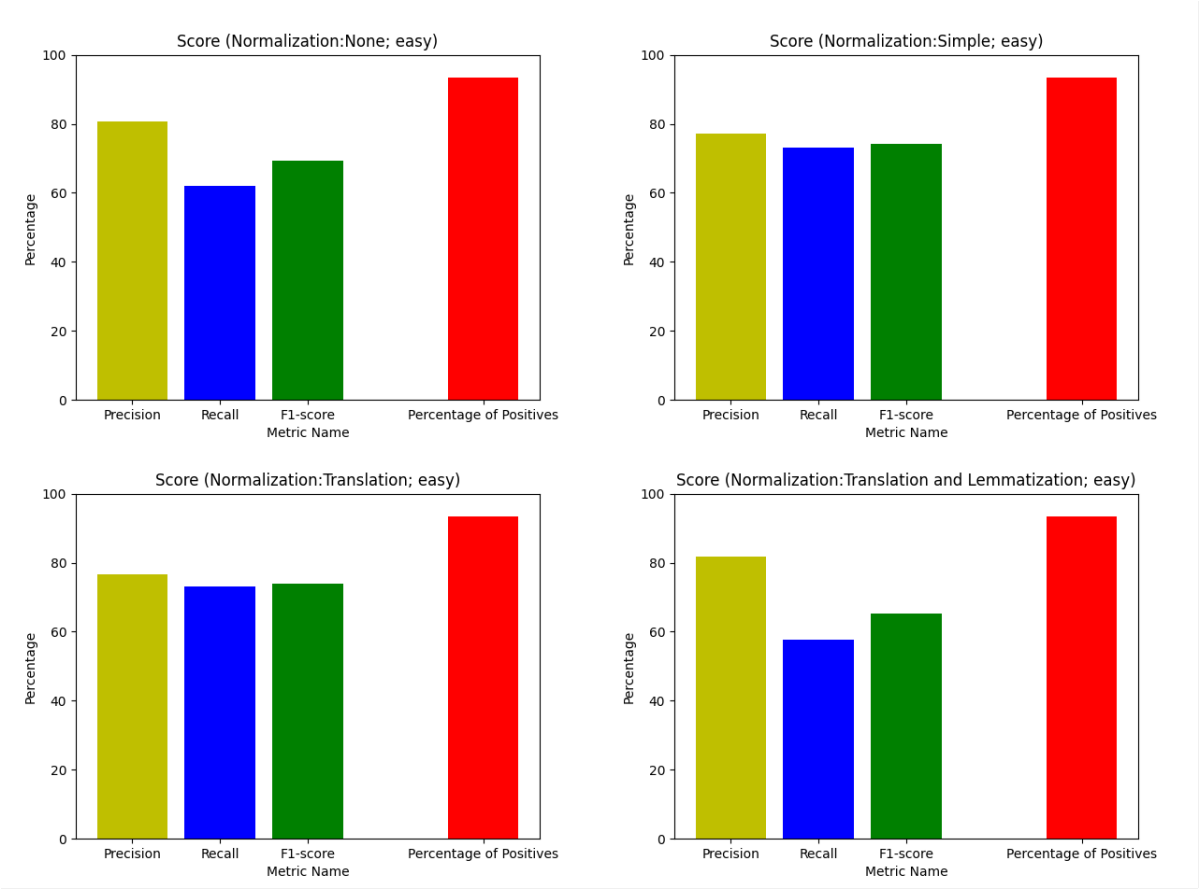

Figure 7.18: Scores for none, simple normalization, normalization with translation and normalization with translation and lemmatization (*Easy Cases*)

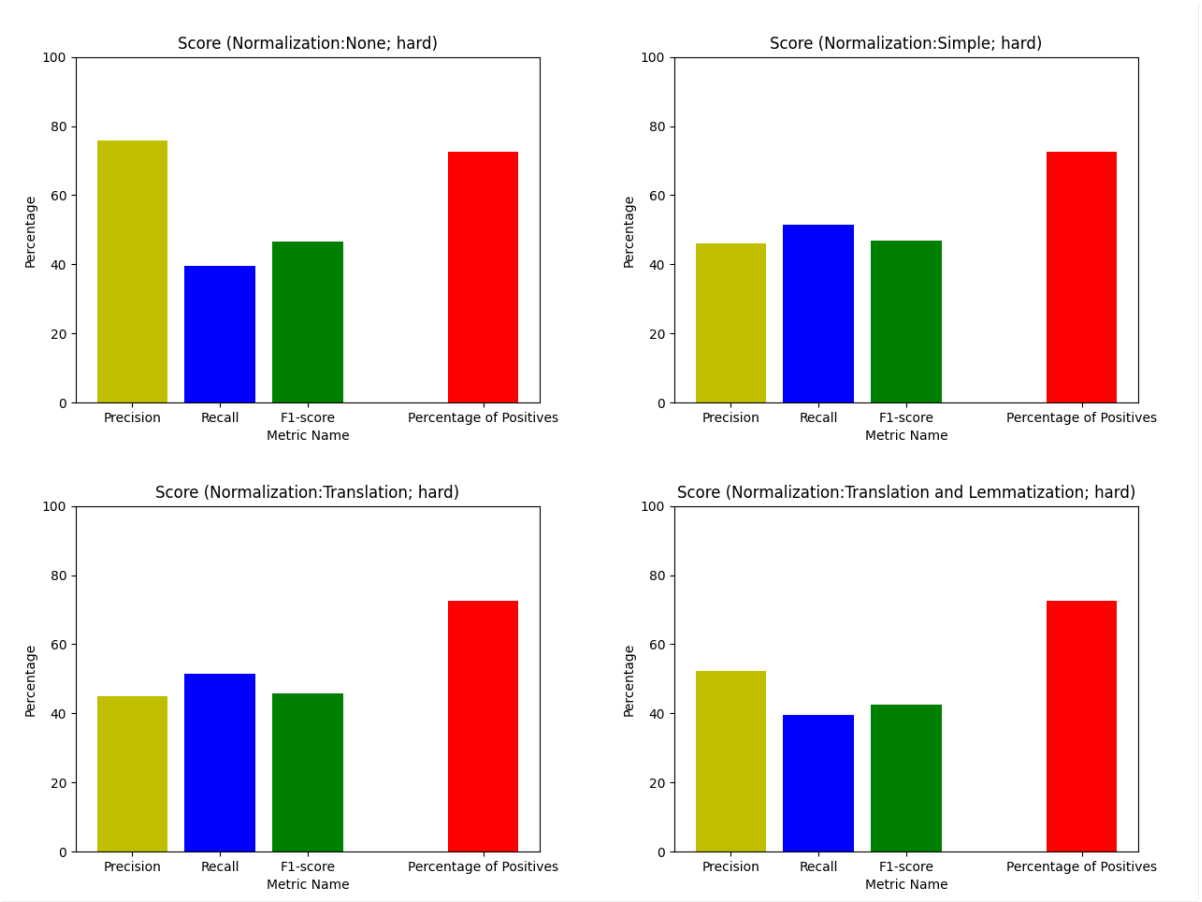

Figure 7.19: Scores for none, simple normalization, normalization with translation and normalization with translation and lemmatization (*Hard Cases*)

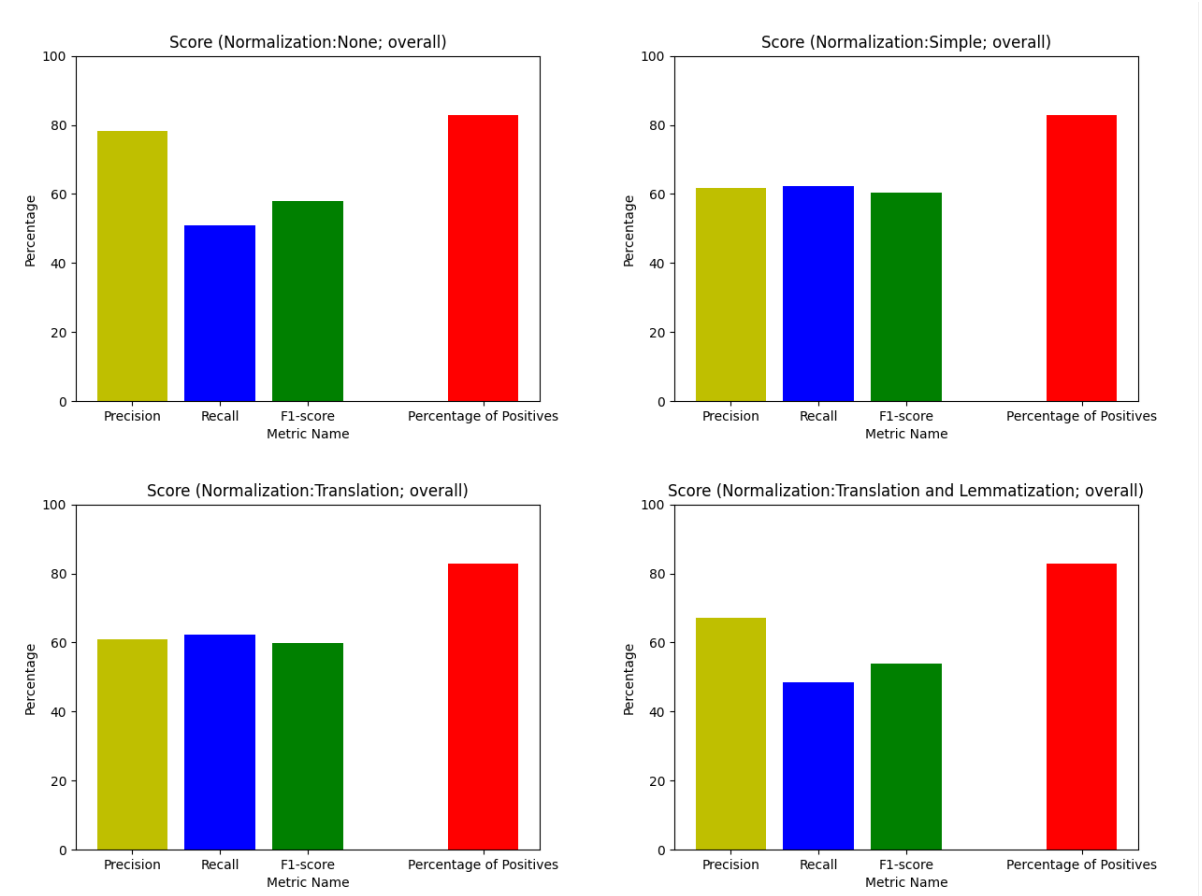

Figure 7.20: Scores for none, simple normalization, normalization with translation and normalization with translation and lemmatization (overall)

Method	Mean Precision (%)	Mean Recall (%)	Mean F1-score (%)	Mean Runtime (min.)
Easy Cases (93.37% of matching attributes in average)				
None	80.77	62.04	69.25	0.09
Simple	77.17	72.98	74.25	0.09
Translation	76.62	72.98	73.99	0.27
Translation and Lemmatization	81.8	57.57	65.29	0.42
Hard Cases (72.48% of matching attributes in average)				
None	75.7	39.63	46.69	0.23
Simple	46.09	51.48	46.78	0.23
Translation	45.07	51.48	45.91	0.7
Translation and Lemmatization	52.31	39.55	42.46	2.3
Overall (82.93% of matching attributes in average)				
None	78.23	50.84	57.97	0.16
Simple	61.63	62.23	60.51	0.16
Translation	60.84	62.23	59.95	0.49
Translation and Lemmatization	67.05	48.56	53.87	1.36

Table 7.5: Sum up of the normalization experiment

The three possibilities of normalization, as well as the absence of it, were tested. The results were not conclusive, since there was not a trend in both Easy Cases (Figure 7.18) and Hard Cases (Figure 7.19) towards a type of normalization. For the Hard Cases, the simple normalization and the non normalization got similar f1-scores, while in the Easy Cases the f1-scores for the simple normalization and the normalization with translation were side-by-side (Table 7.5). In addition, by directly observing a few examples, some outliers and timeouts were spotted. Therefore, it was decided to go deeper and verify if the whole dataset behaved like that.

The normalization tests were executed one more time, but this time focusing on the runtime. Each file in the dataset was repeated five times, calculating afterwards the mean of the runtime.

Boxplots were built in order to evaluate the dispersion of mean runtimes according to the chosen normalization method. There is a huge dispersion on the runtimes for the cases in which a translation is involved (Figure 7.21). Additionally, several outliers exceeding the established five minute runtime limit were found for the mean of the five executions in several files of the dataset (Table 7.6), which demonstrates the translator's inconsistency. Such thing happens because the translator API is an external resource which not only depends on a third party entity, but also needs internet to work.

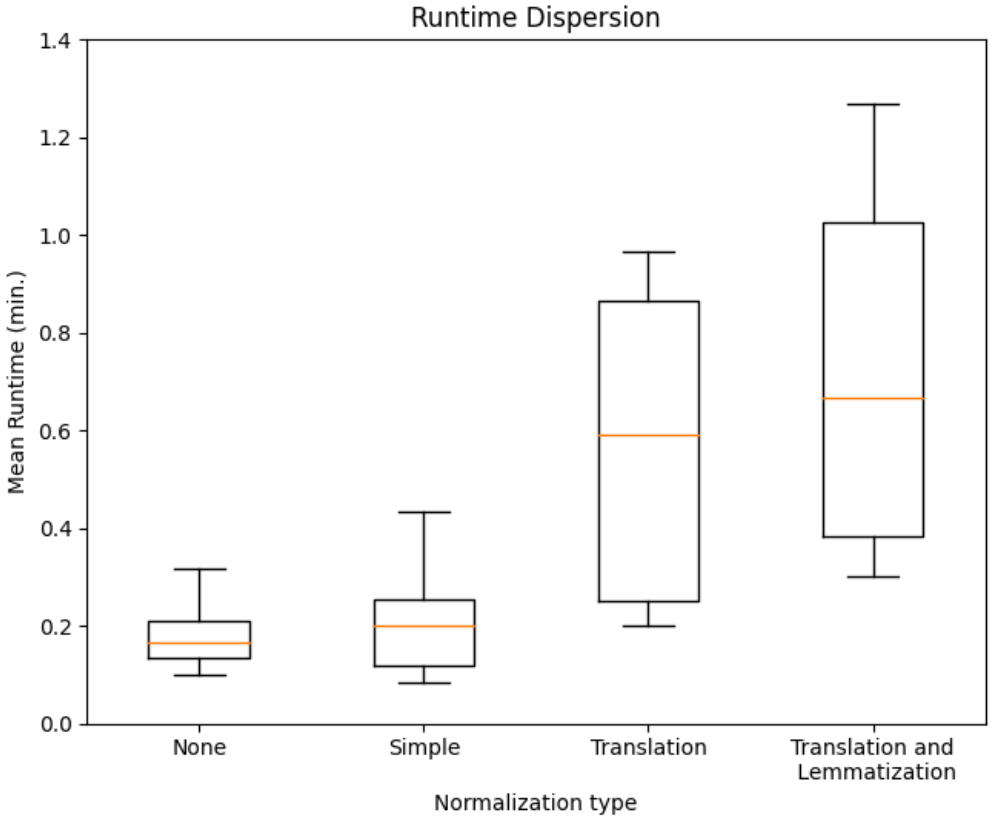

Figure 7.21: Runtime dispersion according to the normalization method

Having into account that it is difficult to quantify the percentage gain between each configuration because the f1-scores were all very similar, and that problems might arise when calling the API, a benefit-cost analysis was made to make sure which normalization method was more worth to apply. The formula to calculate the benefit-cost ratio is:

Method	Number of Outliers	Outliers (min.)
None	1	0.4
Simple	0	—
Translation	4	29.8, 10.967, 3.267, 3.917
Translation and Lemmatization	1	22.783

Table 7.6: Outliers for the normalization experiment (mean for five executions per file)

$$R_{benefit-cost} = \frac{f1}{rt},$$

where $f1$ is the f1-score obtained for a specific file and rt is the average runtime that it takes to match that file. The higher the ratio $R_{benefit-cost}$ value, the better. The mean results Easy Cases, Hard Cases and overall mean can be found in Table 7.7. For the Easy Cases, the best is to apply a simple normalization and for the Hard cases, the absence of it. However, in general, a simple normalization provides the best results, even if it is not by a large margin.

Method	Benefit-cost
Easy Cases	
None	539.041
Simple	606.557
Translation	267.570
Translation and Lemmatization	177.562
Hard Cases	
None	220.680
Simple	188.634
Translation	48.538
Translation and Lemmatization	39.298
Overall	
None	379.860
Simple	397.596
Translation	158.054
Translation and Lemmatization	108.430

Table 7.7: Benefit-cost ratio for each normalization method

It is important to understand the reason for these bad results when translating and lemmatizing. Translating has its advantages, but this translator was not viable in many situations, given the lack of consistency in the API responses and also due to out of context translations (e.g. in Portuguese, *estação* (meteorology) $\longrightarrow$ *station* or *estação* (summer, winter, autumn, spring) $\longrightarrow$ *season*). Speaking of context, lemmatization suffers of similar issues. Lemmatization does not identify the part of speech by itself, which may introduce confusing results for words that assume different parts of speech (e.g. *saw* (noun) $\longrightarrow$ *saw* and *saw* (verb) $\longrightarrow$ *see*).

After the system is all tuned, it is possible to verify which stages of the matching pipeline take longer. The diagram pictured in Figure 7.22, helps visualizing that candidate selection and the matchers running in parallel are the stages that take longer. On the other hand,

parsing the sources, combining matrices and returning the final mapping corresponds to a tiny fraction of the time spent. This makes sense since those are the phases where most computational effort takes place.

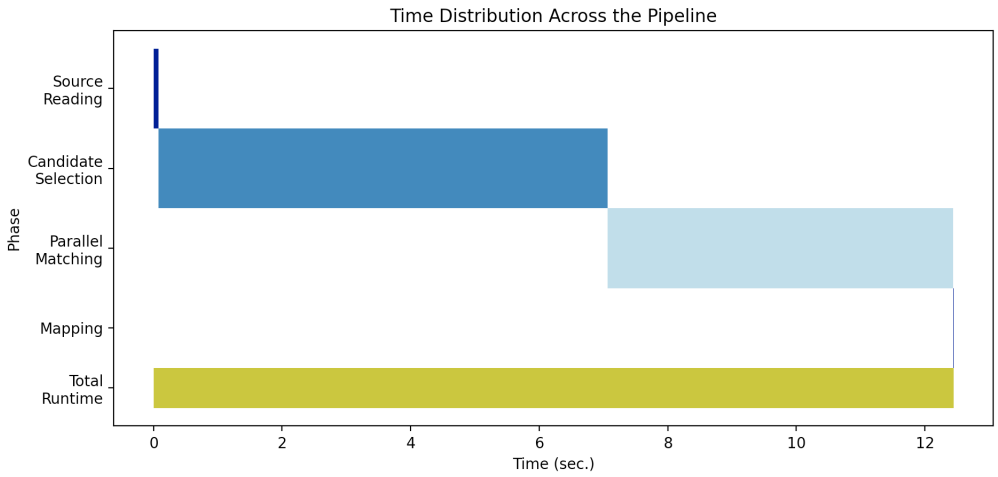

Figure 7.22: Time distribution across different phases of the system's pipeline

7.6 Conclusion

This chapter was focused on the execution of a set of tests whose main goal was to validate and tune the solution. The presentation of the experiment's setup and dataset used for it were followed by the description of three small tests on threshold, parallel matchers and normalization, together with their respective results.

The first test showed the ideal threshold stays bellow 0.50, because the difference between matching and non-matching pairs is very subtle, with only small nuances separating positives from negatives. When testing parallel matchers, it was concluded that both small grams and Levenshtein's measure would work better, due to the attributes' characteristics to which the system is applied. Finally, the tests on nomalization proved that simple normalization is the most effective, at least until a better translator is found.

A final configuration of threshold 0.375, bigrams, Levenshtein measure and a simple normalization was set to be the ideal one found. The new configuration brought an equilibrium between precision and recall, as well as a clear improvement to the scores, reaching an overall mean f1-score of almost 61%.

As expected, *Hard Cases* get lower scores when comparing to the *Easy Cases*, because the schemas to be matched are larger, more complex and the percentage of attributes that have a correspondent is smaller.

This study reinforces the idea that there is always room for improvement, especially on the stages that take longer and are more flexible to customization, like candidate selection and parallel matching. Such observations are next chapter's matter of subject.

Chapter 8

Conclusions and Future Work

This internship was focused on finding an answer to the question "How can we generalize a single platform to work properly for every city?", formulated in Chapter 1, in the context of Ubiwhere's product, UBP.

First off, the problem needed to be defined. That included the formal definition of the problem and understanding in which components of the UBP where it was possible to operate and that could be changed without compromising the current system's pipeline. It was soon realized that this was the application of similar schema and ontology matching problems already defined in the literature. Schema/ontology matching as a research field appeared in the context of an attempt to integrate different sources across the web, as other attempts of standardizing the web, failed.

From an early stage of this project, the challenges associated with such a task were recognized. Among them were the handling of different file formats, structure and entity recognition, attribute matching and validation of the solution.

After limiting the scope of the project, a survey was done to gather information regarding the techniques and tools developed so far. The taxonomy introduced by Rahm and Bernstein, was taken as guideline to categorize the different techniques.

Schema/ontology matching as a research field was very active in the first decade of the century, but has suffered a slowdown since then. Even so, it is possible to watch recent resurgences, motivated by advancements in other areas, such as deep learning, which can be transposed to schema/ontology matching. The study of the state of the art, showed that not all the most advanced machine learning techniques suited the project right now, because the data provided was not enough to train an independent model. Despite these limitations, traditional methods could be successfully used, as their reputation among recent systems was still good.

Based on the acquired knowledge, together with the stakeholders, the requirements were defined, so their interest would be covered.

Development followed an Agile-based methodology and all tasks were organized in a Kanban table according to the project's needs. GitLab was the prime project management tool, as it allowed an easier supervision of the project that was being developed.

Implementation materialized the requirements in light of the proposed architecture and the language chosen was Python. The developed solution was an hybrid system made of several modules with different functions. A module to read and convert the sources, one for extracting candidates using a pre-trained sentence embeddings model, one with several

matchers running in parallel that explore different attribute features, one for extracting the final mapping, and two modules to support the main pipeline, being them a normalization module, equipped with normalization functions and a statistics module, for aiding the system's validation. The produced mappings reach a global cardinality of $? : *$.

There are a few considerations to be made before conceiving an identical system. When matching large schemas where not all attributes have a correspondent, it is advisable to reduce the search space. In this situation, the candidate extraction using embeddings was a viable solution to reduce the number of comparisons and rank the pairs by levels of matching probability. It doubled as a simple structural analyser, as it encoded the whole path from root to the attribute, minimizing the big structural heterogeneity among different cases. Additionally, a modular architecture is vital for the customizing and testing the system.

In order to test the system, an experimental study was conducted focusing mainly on certain configurable parameters, with a perspective of validating and improving the performance of the solution. The dataset built, was split in two. A first half (ten files) called *Easy Cases* and a second half, *Hard Cases*, as a way of including the different typologies of matching happening in the UBP. The experiments demonstrated that it is the threshold which impacts the final score the most, but tuning the other parameters contributes for overall improved scores, too. It also showed that simple normalization pays off over other more complex normalization approaches.

Generally, the project fulfilled its goals, expressed by top priority requirements, and all knowledge acquired for the company. However, it showed to be the early stages of a much bigger project, whose goal is to automatically integrate new data sources into the UBP. Thus, it became clear that future iterations will be required, so that the system is ready to be put into production.

As far as future work is concerned, priority should be given to the assembly of a new dataset. The new dataset has to include much more cases, as well as an extensive diversity of other formats, besides JSON and CSV. Moreover, it is crucial that extra real life files are added to it. A larger and improved dataset will open doors for training models for this specific domain and for employing other advanced machine learning techniques exploring instances that could possibly bring better performances. Furthermore, it will also be feasible to test quality attributes such as scalability, otherwise impossible.

Regarding the system's performance, the f1-score obtained for the *Hard Cases* is still too low to be considered viable to apply this solution to them, as made clear by the tests. This is due to the high schema heterogeneity, nonexistent structure, as it is for the CSV files, mappings to several tables at a time, and matches that are unnatural, for instance id mapping to $external\_id$ instead of Id. Performance improvements could be divided into two parts. On a first phase, the scores could be increased by making upgrades to the already used techniques, for example the normalization method and candidate selection. On a second phase, introducing structural techniques or others exploring new parts of the schema, to enhance even more the true matches. The use of reinforcement learning would also be important, so that the system could identify the tricky mappings, like the one described above, but also to constantly learn even after being deployed.

Finally, after raising the performance, one can think of solving the problem of selecting automatically tables from the entire database that correspond to the schema being matched. Clustering approaches would certainly be a good fit for this problem.

It was mentioned several times in this document that Schema/ontology matching is a

complex and expensive task, only solved heuristically and, consequently, there is always room for improvements. Nonetheless, systems also have to mold themselves to keep up with the integration necessities. The growth in popularity of JSON, fueled by the use of REST APIs, and the development of machine learning technologies should be in the sights of developers that will be in charge of future versions of the system.

This page is intentionally left blank.

References

[1] Airflow proposal. https://cwiki.apache.org/confluence/display/incubator/AirflowProposal. Accessed: 2022-01-13.

[2] Apache Airflow. https://blogs.apache.org/foundation/entry/the-apache-software-foundation-announces44. Accessed: 2022-01-13.

[3] COMA 3.0. https://dbs.uni-leipzig.de/Research/coma.html. Accessed: 2022-01-15.

[4] Django celery. https://docs.celeryproject.org/projects/django-celery/en/2.4/introduction.html. Accessed: 2022-01-13.

[5] Global goals for sustainable development. http://www.globalgoals.org. Accessed: 2021-12-17.

[6] Good ontologies. https://www.w3.org/wiki/Good_Ontologies. Accessed: 2021-12-30.

[7] How LA used big data to build a smart city in the 1970s. https://architexturez.net/pst/az-cf-169297-1435054977. Accessed: 2021-12-16.

[8] Identifying program risks. https://resources.sei.cmu.edu/asset_files/Webinar/2008_018_101_22190.pdf. Accessed: 2022-01-22.

[9] Learn RDF: RDF vs XML. https://cambridgesemantics.com/blog/semantic-university/learn-rdf/rdf-vs-xml/. Accessed: 2021-12-29.

[10] Moscow prioritisation. https://www.agilebusiness.org/page/ProjectFramework_10_MoSCoWPrioritisation? Accessed: 2022-08-04.

[11] Ontology matching. http://www.ontologymatching.org/index.html. Accessed: 2022-01-03.

[12] Ontology matching in biomedical domain. http://disi.unitn.it/~pavel/OM/articles/SWAT4HCLS_Tutorial.pdf. Accessed: 2022-01-05.

[13] Primer example Turtle. https://www.w3.org/2007/OWL/wiki/PrimerExampleTurtle. Accessed: 2021-12-31.

[14] RDF and JSON-LD use cases. https://www.w3.org/2013/dwbp/wiki/RDF_AND_JSON-LD_UseCases. Accessed: 2021-12-31.

[15] Semantic federation of musical and music-related information for establishing a personal music knowledge base - Scientific figure on ResearchGate. https://www.researchgate.net/figure/The-common-layered-Semantic-Web-technology-stack-a-modification-of-Now09-see-also_fig3_215576487. Accessed: 2021-12-29.

[16] Semantic web. https://www.w3.org/2000/Talks/0906-xmlweb-tbl/text.htm. Accessed: 2021-12-26.

[17] Semantic web. https://devopedia.org/semantic-web. Accessed: 2021-12-26.

[18] Semantic web standards. https://joinup.ec.europa.eu/sites/default/files/inline-files/W3C02.pdf. Accessed: 2021-12-28.

[19] Tesaurus. https://www.oxfordlearnersdictionaries.com/definition/english/thesaurus?q=thesaurus. Accessed: 2022-01-14.

[20] Turtle. https://www.w3.org/TeamSubmission/turtle/. Accessed: 2021-12-30.

[21] Turtle examples. https://www.loc.gov/aba/pcc/bibco/documents/TurtleExamples.pdf. Accessed: 2021-12-31.

[22] Ubiwhere urban platform. https://www.ubiwhere.com/en/products/smart-cities/urban-platform. Accessed: 2021-12-16.

[23] Understanding linked data formats. https://medium.com/wallscope/understanding-linked-data-formats-rdf-xml-vs-turtle-vs-n-triples-eb931dbe9827. Accessed: 2021-12-30.

[24] United Nations revision of world urbanization prospects 2018. https://www.un.org/development/desa/en/news/population/2018-revision-of-world-urbanization-prospects.html. Accessed: 2021-12-15.

[25] Urban platform. https://urbanplatform.city. Accessed: 2021-12-16.

[26] What is RDF? https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf/. Accessed: 2021-12-29.

[27] What is the difference between RDF and OWL? https://stackoverflow.com/questions/1740341/what-is-the-difference-between-rdf-and-owl, note = Accessed: 2021-12-31.

[28] Alsayed Algergawy, Eike Schallehn, and Gunter Saake. Understanding the schema matching problem. 01 2007.

[29] Ali A. Alwan, Azlin Nordin, Mogahed Alzeber, and Abedallah Zaid Abualkishik. A survey of schema matching research using database schemas and instances. *International Journal of Advanced Computer Science and Applications*, 8(10), 2017.

[30] Ahmad Assaf, Eldad Louw, Aline Senart, Corentin Follenfant, Raphaël Troncy, and David Trastour. Improving schema matching with linked data. https://arxiv.org/abs/1205.2691, 2012. Accessed: 2022-05-09.

[31] David Aumueller, Hong Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with coma++. pages 906–908, 01 2005.

[32] Daniel Ayala, Inma Hernández, David Ruiz, and Erhard Rahm. Leapme: Learning-based property matching with embeddings. *Data Knowledge Engineering*, 137:101943, 2022.

[33] Daniel Ayala, Inma Hernández, David Ruiz, and Miguel Toro. Tapon: A two-phase machine learning approach for semantic labelling. *Knowledge-Based Systems*, 163, 11 2018.

[34] Rangaswami Balakrishnan and Kanna Ranganathan. *A textbook of graph theory*. Springer Science & Business Media, 2012.

[35] Nils Barlaug and Jon Atle Gulla. Neural networks for entity matching: A survey. *ACM Transactions on Knowledge Discovery from Data*, 15(3):1–37, 2021.

[36] Alexander Beider and Stephen Morse. Phonetic matching: A better soundex. `https://stevemorse.org/phonetics/bmpm2.pdf`, 2010. Accessed: 2022-04-20.

[37] Zohra Bellahsene, Angela Bonifati, Fabien Duchateau, and Yannis Velegrakis. *On Evaluating Schema Matching and Mapping*, pages 253–291. 12 2011.

[38] Zohra Bellahsene, Angela Bonifati, and Erhard Rahm. *Schema Matching and Mapping*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[39] Samy Bengio and Georg Heigold. Word embeddings for speech recognition. In *Proceedings of the 15th Conference of the International Speech Communication Association, Interspeech*, 2014.

[40] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *ScientificAmerican.com*, 05 2001.

[41] Philip Bernstein, Jayant Madhavan, and Erhard Rahm. Generic schema matching, ten years later. *PVLDB*, 4:695–701, 08 2011.

[42] Philip A. Bernstein and Sergey Melnik. Model management 2.0: Manipulating richer mappings. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 1–12. Association for Computing Machinery, 2007.

[43] Vitor Marini Blaselbauer and Joao Marcelo Borovina Josko. JSONGlue: A hybrid matcher for JSON schema matching. In *Proceedings of the Brazilian Symposium on Databases*, 2020.

[44] Willem Nico Borst and W.N. Borst. *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. PhD thesis, University of Twente, Netherlands, September 1997.

[45] Nadjet Bouayad-Agha. Sentence similarity with bert vs sbert. `https://github.com/nadjet/sentence_similarity`. Accessed: 2022-04-20.

[46] James Briggs. Bert for measuring text similarity. `https://towardsdatascience.com/bert-for-measuring-text-similarity-eec91c6bf9e1`. Accessed: 2022-04-20.

[47] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. Creating embeddings of heterogeneous relational datasets for data integration tasks. pages 1335–1349, 06 2020.

[48] Andrew Carstairs-McCarthy. *An Introduction to English Morphology: Words and Their Structure (2nd edition)*. Edinburgh University Press, 2018.

[49] Peter Christen. A comparison of personal name matching: Techniques and practical issues. *in 'The Second International Workshop on Mining Complex Data (MCD'06)*, 12 2006.

[50] Rudi Cilibrasi and Paul Vitányi. The Google similarity distance. *Knowledge and Data Engineering, IEEE Transactions on*, 19:370–383, 04 2007.

[51] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *IJPRAI*, 18:265–298, 05 2004.

[52] Orsolya Csiszar. Ordered weighted averaging operators: A short review. *IEEE Systems, Man, and Cybernetics Magazine*, 7(2):4–12, 2021.

[53] Marcin Detyniecki. Fundamentals on aggregation operators. *Proceedings AGOP, Asturias, Spain*, 01 2001.

[54] R. P. Devi and Dr. V. Thigarasu. A novel approach for record deduplication using hidden markov model. 2014.

[55] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. `https://arxiv.org/abs/1810.04805`, 2018. Accessed: 2022-04-10.

[56] Robin Dhamankar, Yoonkyong Lee, Anhai Doan, Alon Halevy, and Pedro Domingos. imap: Discovering complex semantic matches between database schemas. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 04 2004.

[57] Giorgio Maria Di Nunzio and Federica Vezzani. A linguistic failure analysis of classification of medical publications: A study on stemming vs lemmatization. 01 2018.

[58] Hong Do and Erhard Rahm. COMA - A system for flexible combination of schema matching approaches. pages 610–621, 08 2002.

[59] AnHai Doan, Pedro M. Domingos, and Alon Y. Levy. Learning source descriptions for data integration. 2000.

[60] AnHai Doan, Alon Halevy, and Zachary G Ives. *Principles of data integration*. Morgan Kaufmann, 2012.

[61] AnHai Doan, Jayant Madhavan, Pedro Domingos, and Alon Halevy. Learning to map between ontologies on the semantic web. page 662, 01 2002.

[62] Irvin Dongo, Firas Al Khalil, Richard Chbeir, and Yudith Cardinale. Semantic web datatype similarity: Towards better rdf document matching. In Djamal Benslimane, Ernesto Damiani, William I. Grosky, Abdelkader Hameurlain, Amit Sheth, and Roland R. Wagner, editors, *Database and Expert Systems Applications*, pages 189–205, Cham, 2017. Springer International Publishing.

[63] Marc Ehrig, Steffen Staab, and York Sure. Bootstrapping ontology alignment methods with apfel. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, page 1148–1149, New York, NY, USA, 2005. Association for Computing Machinery.

[64] Jérôme Euzenat, Christian Meilicke, Pavel Shvaiko, Heiner Stuckenschmidt, and Cassia Trojahn dos Santos. Ontology Alignment Evaluation Initiative: Six years of experience. *Journal on Data Semantics*, XV(6720):158–192, 2011. euzenat2011b.

[65] Jérôme Euzenat and Pavel Shvaiko. *Ontology matching: Second edition*. 10 2013.

[66] Hugging Face. all-minilm-l6-v2. `https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2`. Accessed: 2022-06-10.

[67] Hongjie Fan, Junfei Liu, and Kejun Deng. Towards a composite xml schema matching approach using reference ontology. In *2016 3rd International Conference on Information Science and Control Engineering (ICISCE)*, pages 724–728, 2016.

[68] Daniele Varrazzo Federico Di Gregorio. Adaptation of python values to sql types. `https://www.psycopg.org/docs/usage.html#adapt-numbers`. Accessed: 2022-08-05.

[69] Apache Software Foundation. Phonetic matching. `https://solr.apache.org/guide/7_4/phonetic-matching.html#beider-morse-phonetic-matching-bmpm`, 2018. Accessed: 2022-04-20.

[70] Philippe Fournier-Viger. The semantic web and why it failed. `https://data-mining.philippe-fournier-viger.com/the-semantic-web-and-why-it-failed/`, 2018.

[71] Andrea Frome, Greg Corrado, Jonathon Shlens, Samy Bengio, Jeffrey Dean, Marc'Aurelio Ranzato, and Tomas Mikolov. Devise: A deep visual-semantic embedding model. In *Neural Information Processing Systems (NIPS)*, 2013.

[72] T.N. Gadd. Phonix: The algorithm. *Program 24*, pages 363–366, 1990.

[73] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[74] Anselmo Guedes, Fernanda Baião, and Kate Revoredo. On the identification and representation of ontology correspondence antipatterns. *CEUR Workshop Proceedings*, 1248, 01 2014.

[75] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[76] Do Hong Hai. *Schema matching and mapping-based data integration.* PhD thesis, University of Leipzig, 2005.

[77] Daniel Ayala Hernández, Inma Hernández, David Ruiz, and Erhard Rahm. Towards the smart use of embedding and instance features for property matching. *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2111–2116, 2021.

[78] Tran Hong-Minh and Dan Smith. Hierarchical approach for datatype matching in xml schemas. In *24th British National Conference on Databases (BNCOD'07)*, pages 120–129, 2007.

[79] Ian Horrocks, Bijan Parsia, Peter Patel-Schneider, and James Hendler. Semantic web architecture: Stack or two towers? pages 37–41, 09 2005.

[80] Dan Hughes. Non-functional requirement examples. `https://wittij.com/non-functional-requirement-examples/`. Accessed: 2022-07-30.

[81] Benjamin Hättasch, Michael Truong-Ngoc, Andreas Schmidt, and Carsten Binnig. It's ai match: A two-step approach for schema matching using embeddings. `https://arxiv.org/abs/2203.04366`, 2022. Accessed: 2022-05-07.

[82] Andra Ionescu. Reproducing state-of-the-art schema matching algorithms. Master's thesis, Delft University of Technology, 2020.

[83] Dan Jurafsky and James H Martin. Speech and language processing (3rd draft ed.), 2019. Accessed: 2022-03-20.

[84] Yannis Kalfoglou and Marco Schorlemmer. Ontology mapping: The state of the art. 2, 01 2003.

[85] Anastasios Kementsietsidis. Schema matching. *Encyclopedia of Database Systems*, 2009.

[86] Mohamed Kettouch, Cristina Luca, and Mike Hobbs. Schema matching for semi-structured and linked data. In *2017 IEEE 11th International Conference on Semantic Computing (ICSC)*, pages 270–271, 2017.

[87] Mahdi Khalaf, Hayder Najm, Alaa Abdulhussein Daleh, Ali Hasan Munef, and Ghaith Mojib. Schema matching using word-level clustering for integrating universities' courses. In *2020 2nd Al-Noor International Conference for Science and Technology (NICST)*, pages 1–6, 2020.

[88] Grzegorz Kondrak. N-gram similarity and distance. SPIRE'05, page 115–126, Berlin, Heidelberg, 2005. Springer-Verlag.

[89] Saketh Kotamraju. An intuitive explanation of sentence-bert. `https://towardsdatascience.com/an-intuitive-explanation-of-sentence-bert-1984d144a868`. Accessed: 2022-04-20.

[90] Saruladha Krishnamurthy, Aghila Gnanasekaran, and B Sathiya. A comparative analysis of ontology and schema matching systems. *International Journal of Computer Applications*, 34, 01 2011.

[91] Markus Krötzsch, František Simančík, and Ian Horrocks. A description logic primer. *CoRR*, abs/1201.4089, 2012.

[92] J. A. Larson, S. B. Navathe, and R. Elmasri. A theory of attributed equivalence in databases with application to schema integration. *IEEE Trans. Softw. Eng.*, 15(4):449–463, apr 1989.

[93] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, and Sören et al. Auer. DBpedia – A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[94] Wen-Syan Li and Chris Clifton. SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data Knowl. Eng.*, 33:49–84, 04 2000.

[95] Dekang Lin. An information-theoretic definition of similarity. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, page 296–304, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[96] Julie Beth Lovins. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11:22–31, 1968.

[97] Fhabiana dos Santos Machado. Um processo para extração de esquemas conceituais em fontes de dados json baseado em técnicas de similaridade de texto. Master's thesis, Universidade Federal de Santa Maria, 2017.

[98] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, page 49–58. Morgan Kaufmann Publishers Inc., 2001.

[99] Ahmed Mahdi and Sabrina Tiun. Utilizing wordnet and regular expressions for instance-based schema matching. *Research Journal of Applied Sciences, Engineering and Technology*, 8, 07 2014.

[100] Anan Marie and Avigdor Gal. A.: Boosting schema matchers. volume 5331, pages 283–300, 11 2008.

[101] Sabine Maßmann, Salvatore Raunich, David Aumüller, Patrick Arnold, and Erhard Rahm. Evolution of the coma match system. volume 814, 01 2011.

[102] Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. A short history of schema mapping systems. *Proceedings of the 20th Italian Symposium on Advanced Database Systems, SEBD 2012*, pages 99–106, 01 2012.

[103] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. `https://arxiv.org/abs/1301.3781`, 2013. Accessed: 2022-04-05.

[104] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. `https://arxiv.org/abs/1310.4546`, 2013. Accessed: 2022-04-05.

[105] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, nov 1995.

[106] Renée Miller, Mauricio Hernández, Laura Haas, Ling-Ling Yan, C. Ho, Ronald Fagin, and Lucian Popa. The clio project: Managing heterogeneity. *SIGMOD Record*, 30:78–83, 03 2001.

[107] Renata Junges Padilha. Um processo para casamento de esquemas de documentos json baseado na estrutura e nas instâncias. Master's thesis, Universidade Federal de Santa Maria, 2020.

[108] Sid Panjwani. Multiprocessing vs. threading in python: What you need to know. `https://timber.io/blog/multiprocessing-vs-multithreading-in-python-what-you-need-to-know/`. Accessed: 2022-05-23.

[109] Jeffrey Partyka, Latifur Khan, and Bhavani Thuraisingham. Semantic schema matching without shared instances. In *2009 IEEE International Conference on Semantic Computing*, pages 297–302, 2009.

[110] Ashok Malhotra Paul V. Biron. Xml schema part 2: Datatypes second edition. `https://www.w3.org/TR/xmlschema-2/`. Accessed: 2022-08-05.

[111] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. volume 14, pages 1532–1543, 01 2014.

[112] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. `https://arxiv.org/abs/1802.05365`, 2018. Accessed: 2022-04-10.

[113] Eric Peukert, Sabine Maßmann, and Kathleen König. Comparing similarity combination methods for schema matching. pages 692–701, 08 2010.

[114] Martin F Porter. An algorithm for suffix stripping. *Program*, 1980.

[115] Martin F. Porter. Snowball: A language for stemming algorithms. 2001.

[116] Jan Portisch, Michaela Hladik, and Heiko Paulheim. Background knowledge in schema matching: Strategy vs. data. In *SEMWEB*, 2021.

[117] Mateusz Przyborowski, Mateusz Pabiś, Andrzej Janusz, and Dominik Ślęzak. Schema matching using gaussian mixture models with wasserstein distance. 11 2021.

[118] Erhard Rahm and Phil Bernstein. On matching schemas automatically. Technical Report MSR-TR-2001-17, February 2001.

[119] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[120] Thomas Rebele, Fabian Suchanek, Johannes Hoffart, Joanna Biega, Erdal Kuzey, and Gerhard Weikum. YAGO: A multilingual knowledge base from Wikipedia, Wordnet, and Geonames. pages 177–185, 10 2016.

[121] Nils Reimers. Pretrained models. `https://www.sbert.net/docs/pretrained_models.html`. Accessed: 2022-06-10.

[122] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *CoRR*, abs/1908.10084, 2019.

[123] Philip Resnik. Using information content to evaluate semantic similarity in a taxonomy. `https://arxiv.org/abs/cmp-lg/9511007`, 1995. Accessed: 2022-05-07.

[124] Diego Rodrigues and Altigran da Silva. A study on machine learning techniques for the schema matching network problem. `https://journal-bcs.springeropen.com/articles/10.1186/s13173-021-00119-5#citeas`, 2021. Accessed: 2022-04-27.

[125] Xin Rong. word2vec parameter learning explained. `https://arxiv.org/abs/1411.2738`, 2014. Accessed: 2022-04-05.

[126] Thomas Saaty. Decision making with the analytic hierarchy process. *Int. J. Services Sciences Int. J. Services Sciences*, 1:83–98, 01 2008.

[127] Harald Sack. Linked data engineering. `https://www.youtube.com/watch?v=BbcD-ah8dtE&list=PLoOmvuyo5UAfY6jb46jCpMoqb-dbVewxg&index=19`, 2016.

[128] Seva Safris. A deep look at json vs. xml, part 1: The history of each standard. `https://www.toptal.com/web/json-vs-xml-part-1`.

[129] Khalid Saleem and Zohra Bellahsene. New challenges in data integration: Large scale automatic schema matching. 04 2007.

[130] Helmut Schmid. Improvements in part-of-speech tagging with an application to german. 1999.

[131] Helmut Schmidt. Probabilistic part-of-speech tagging using decision trees. 1994.

[132] Khurram Shahzad, Ifrah Pervaz, Rao Muhammad, and Adeel Nawab. Wordnet-based semantic similarity measures for process model matching. 09 2018.

[133] Sheetal Sharma, Darothi Sarkar, and Divya Gupta. Agile processes and methodologies: A conceptual study. *International Journal on Computer Science and Engineering*, 4, 05 2012.

[134] Manjula Shenoy, Karthish Shet, and Dinesh Acharya. A new similarity measure for taxonomy based on edge counting. 11 2012.

[135] Amit P. Sheth. Changing focus on interoperability in information systems:from system, syntax, structure to semantics. *Interoperating Geographic Information Systems*, 1999.

[136] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. In *Journal on data semantics IV*, pages 146–171. Springer, 2005.

[137] Pavel Shvaiko and Jérôme Euzenat. Tutorial on schema and ontology matching. `http://dit.unitn.it/~accord/Presentations/ESWC%2705-MatchingHandOuts.pdf`. Accessed: 2022-01-13.

[138] António Silva. How to write meaningful quality attributes for software development. `https://www.codementor.io/@antoniopfesilva/how-to-write-meaningful-quality-%20attributes-for-software-development-ez8y90wyo`.

[139] Peter Simons. Ontology. `https://www.britannica.com/topic/ontology-metaphysics`, 2015.

[140] Jasmeet Singh and Vishal Gupta. A systematic review of text stemming techniques. *Artif. Intell. Rev.*, 48(2):157–217, 2017.

[141] John Miles Smith, Philip A. Bernstein, Umeshwar Dayal, Nathan Goodman, Terry Landers, Ken W. T. Lin, and Eugene Wong. Multibase: Integrating heterogeneous distributed database systems. In *Proceedings of the May 4-7, 1981, National Computer Conference*, AFIPS '81, page 487–499. Association for Computing Machinery, 1981.

[142] Raquel Kolitski Stasiu. *Avaliação da qualidade de funções de similaridade no contexto de consultas por abrangência*. PhD thesis, Universidade Federal do Rio Grande do Sul, 2007.

[143] Jeff Sutherland. Why gantt charts were banned in the first scrum `https://www.scruminc.com/why-gantt-charts-were-banned-in-first/`. Accessed: 2022-02-25.

[144] Thulazshini Tamilchelvan. How do gantt charts support agile development? `https://medium.com/project-managers-planet/how-do-gantt-charts-support-agile-development-224f92c9a951`. Accessed: 2022-02-25.

[145] Cássia Trojahn, Renata Vieira, Daniela Schmidt, Adam Pease, and Giancarlo Guizzardi. Foundational ontologies meet ontology matching: A survey. *Semantic Web*, 2021.

[146] Jacco Van Ossenbruggen, Lynda Hardman, and Lloyd Rutledge. Hypermedia and the semantic web: A research agenda. *Journal of Digital Information; Vol 3, No 1 (2002)*, 3, 01 2002.

[147] Achille C. Varzi. On doing ontology without metaphysics. *Philosophical Perspectives*, 25(1):407–423, 2011.

[148] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. `https://arxiv.org/abs/1706.03762`, 2017. Accessed: 2022-04-15.

[149] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, sep 2014.

[150] Yaoshu Wang, Jianbin Qin, and Wei Wang. Efficient approximate entity matching using jaro-winkler distance. pages 231–239, 10 2017.

[151] Tingting Wei, Yonghe Lu, Huiyou Chang, Qiang Zhou, and Xianyu Bao. A semantic approach for text clustering using wordnet and lexical chains. *Expert Systems with Applications*, 42(4):2264–2275, 2015.

[152] Junchi Yan, Xu-Cheng Yin, Weiyao Lin, Cheng Deng, Hongyuan Zha, and Xiaokang Yang. A short survey of recent advances in graph matching. In *Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval*, New York, NY, USA, 2016. Association for Computing Machinery.

[153] ChuanTao Yin, Zhang Xiong, Hui Chen, JingYuan Wang, Daven Cooper, and Bertrand David. A literature survey on smart cities. *Science China Information Sciences*, 58(10), 2015.

[154] Jing Zhang, Bonggun Shin, Jinho D. Choi, and Joyce C. Ho. Smat: An attention-based deep learning solution to the automation of schema matching. In Ladjel Bellatreche, Marlon Dumas, Panagiotis Karras, and Raimundas Matulevičius, editors, *Advances in Databases and Information Systems*, pages 260–274. Springer International Publishing, 2021.

[155] Zhi Zhang, Pengfei Shi, Haoyang Che, and Jun Gu. An algebraic framework for schema matching. *Informatica*, 19:421–446, 2008.

[156] Zhi Zhang, Pengfei Shi, Haoyang Che, and Yong Sun. Formulation schema matching problem for combinatorial optimization problem. *IBIS*, 1:33–60, 01 2006.

[157] Aleksandar Šotić and Radenko Rajić. The review of the definition of risk. *Online Journal of Applied Knowledge Management*, pages 17–26, 2015.