ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

TIME-EVOLVING KNOWLEDGE GRAPHS BASED ON POIROT: DYNAMIC REPRESENTATION OF PATIENTS' VOICES

Elaborato in
Programmazione

Relatore
Prof. Antonella Carbonaro

Co-relatore
Dott. Giacomo Frisoni

Presentata da
Samuele Ceroni

Terza Sessione di Laurea
Anno Accademico 2019 – 2020

KEYWORDS

Knowledge graph
Semantic Web
Natural Language Processing
Dynamic representation
Rare diseases

Dedicated to all those who supported and believed in me,
Even despite the darkness and the distance.

Abstract

Nowadays people are spending more and more time online: this is a permanent change that leads to a huge amount of diversified data like never before which needs to be managed to extrapolate knowledge from it.

This also involves social media which produces free textual information very difficult to process, but occasionally very useful. For instance, in the field of rare diseases, our specific testing context could lead to the possibility to organize the voice of patients and of caregivers, difficult to gather otherwise.

People who are affected by a rare disease often strive to find enough information about it. Indeed, not much material is available online and the number of doctors qualified for those specific diseases is quite limited. Social networks become then the best place to exchange ideas and opinions. The main difficulty in finding useful information on social networks though is that text gets lost quickly and it's not straightforward to give a semantic structure to it and dynamically evolve this representation over time.

In literature, there are some techniques that manage to transform unstructured data into useful information, extracting them using artificial intelligence. These techniques are often well expressive and are able to precisely convert data into knowledge, but they are not directly connected to text sources nor to a system that stores and allows to update the extrapolated information. Consequently, they are not well automated in incrementally keeping information up-to-date as new text is provided, resulting in the need for a mechanical process to do it.

The contribution proposed in this thesis focuses on how to use these technologies to maintain information in order over time, enhancing their usability and freshness. It consists of a system that connects the text source providers to the built knowledge graph, which contains the knowledge acquired and updated.

Introduction

We now live in times where it's common to spend most of our day online. Despite the negative ways this might affects our wellbeing, there's also a positive side on this: a lot of diversified data is available on the web. In fact, people spend hours a day consulting social media, websites, watching videos, sharing experiences through textual posts, images and videos. This has created a parallel world extremely connected and diversified, where billions of people around the world constantly are protagonist and demolished the barrier of distance. In this world people feel more and more free to share their private lives, especially if they find spots where they don't feel judged or taunted, but they feel part of a community where they understand each other. This process doesn't have precedents and, like never before, lot of diversified data is created every day. Nevertheless data itself is useless on it's own if not interpreted and if knowledge is not extrapolated from it. This is a current problem which is subject of research and which commits many experts of the field.

The places where most of free textual information are shared and stored are social medias. Among these data, some of them are occasionally very useful but difficult to process. There are many groups or pages online where a lot of data or instructions are lost over time. It may happen that some of them remain valid over time but they get buried by new ones. There are tons of examples, like social groups of expats looking for information about local policies or regulations, or, in the field of rare diseases, our specific testing context, groups where the "voice of patients" and caregivers emerges. These contributes are very precious and, sometimes, they are unique and not gatherable elsewhere, but in order to be usable they need to be organized.

Indeed it's common that, people who are affected by a disease which is rare by definition, find many difficulties when it comes to find information about it. Firstly there is not much organized material available online to be consulted and the number of doctor specialized into single diseases are quite limited. In this scenario poor of answers, social networks become therefore the best place to exchange ideas and opinions. Another difficulty in finding useful information on social networks is that it's not straightforward to keep them arranged in real-time.

In literature, through the usage of Artificial Intelligence (AI), some techniques exist and have achieved good results in transforming unstructured data into useful information. However, these techniques have encountered obstacles in finding a solution which could:

- be a descriptive text mining (find the reason behind events);
- automatically read and understand natural language;
- find statistically significant correlations;
- explain and interpret correlations in a statistical way, with a low number of dimensions (differently from neural networks);
- be unsupervised;
- be domain independent.

Furthermore, these techniques are often well expressive, but another of their limit is that they are not communicating directly with neither a text source nor to a storage system which allows to update the knowledge. This means that an automation in keeping information up-to-date incrementally is missing and therefore the process to do it is quite mechanical.

This thesis aims to fill this gap by implementing an autonomous solution which uses these technologies to maintain extrapolated knowledge in order over time. It consists of a web application which uses a recent methodology that satisfied all the previous requirements and connects text source provider to the built knowledge graph, which contains the knowledge acquired and updated over time.

The thesis has been divided in the following chapters:

- **Chapter 1** - Background on how knowledge can be represented through the usage of standards, different type of graphs architectures, the world of integration of knowledge worldwide and its evolution over time;
- **Chapter 2** - Context on which the work has been tested, the voice of patients of people affected by a rare disease and the methodology used to extrapolate information from it;
- **Chapter 3** - The comparison of knowledge graph tools that are available to store the created connections and the chosen one for this project;
- **Chapter 4** - The development of the project which keeps a knowledge graph updated from a textual source in real time;
- **Chapter 5** - Conclusions and further developments which would maximize the potential with further implementations.

Contents

1 Background 1

1.1 Knowledge representation . . . 1

1.1.1 Ontologies . . . 1

1.1.2 Knowledge Graphs . . . 2

1.1.3 Ontologies vs Knowledge Graphs . . . 2

1.1.4 Web semantic standards . . . 2

1.2 Graph types . . . 5

1.2.1 Semantic Graphs . . . 6

1.2.2 Labeled Property Graphs . . . 7

1.2.3 Multi-model Graphs . . . 8

1.3 Linked Open Data . . . 9

1.4 Time-Evolving Graph . . . 11

2 Context 13

2.1 The Voice of Patients . . . 13

2.2 POIROT . . . 14

2.2.1 Methodology . . . 14

2.2.2 Application to Knowledge Graph Learning . . . 17

2.2.3 Dataset . . . 19

2.2.4 Limitations . . . 19

2.3 Goals . . . 20

3 Knowledge Graph tools 21

3.1 Overview . . . 21

3.1.1 Protégé . . . 21

3.1.2 Allegrograph . . . 22

3.1.3 Grakn . . . 22

3.1.4 Neo4j . . . 22

3.1.5 Stardog . . . 23

3.2 Comparison with table . . . 24

3.3 Knowledge graph database vs graph database . . . 31

3.4 Chosen tool . . . 31

4 Project **33**
4.1 Web server 35
4.2 Data source 37
4.3 Text to graph triples 40
4.4 Store triples 44

5 Conclusions and further developments **49**

Bibliography **51**

List of Figures

1.1 Graph Types Model Comparison. 6
1.2 Graph Types Technology comparison. 7
1.3 DB engines popularity [1] 8
1.4 RDF vs LPG triple representation 8
1.5 Semanticness of the web through years 10
1.6 LOD cloud 10

2.1 Poirot modularity 15
2.2 LSA matrix decomposition and their reduction through SVD [2] 16
2.3 Learning layer cake [3] 18

4.1 Application architecture 34
4.2 Stardog query before triple insertion. 48
4.3 Stardog query after triple insertion. 48

Chapter 1

Background

1.1 Knowledge representation

Human brains have their way to represent knowledge and how to remember it. Most of the time it varies from person to person and there's no standard to regulate it.
When it comes to informatics, a formal model is needed to represent knowledge and make it understandable by machines. There are several ways to do it, like using Ontologies or Knowledge Graphs.

1.1.1 Ontologies

One way of representing information is through the usage of Ontologies. This term is used to mean *"the study of categories of things that exist or may exist in some domain"* [4]. Indeed, the word Ontology comes from the antique Greek words ontos (being) and logos (study). Ontology doesn't have a universal definition, but the most common in Computer Science has been given by Thomas R. Gruber: *"an ontology is an explicit, formal specification of a shared conceptualization. The term is borrowed from philosophy, where an Ontology is a systematic account of Existence. For AI systems, what 'exists' is that which can be represented"* [5]. This can be better explained exploding the meaning of the keywords: conceptualization refers to the abstract model of the world within the domain considered — and it needs to be shared to capture consensual knowledge; it must be explicit because there shouldn't be any concept left undefined; formal indicates that it must be usable by machines.

An ontology is therefore a way to describe knowledge-creating formal concepts and relations between them. The formalization of those concepts takes place in the following components: class, individuals, and relations. Classes are composed of attributes (name-value pairs) and can be linked to other classes.

Relations are special attributes that link two classes (or more in hypergraphs) and their value are therefore objects of other classes. Individuals are instances of a class and can have relations with other classes or other individuals. For example, `Samuele` can be an individual of the class `Person` and it specializes in it. Classes used in an ontology can also be linked to classes that don't necessarily belong to the same ontology, enabling creating a cross-database search and interoperability. Generalization is as important as the relationship between different classes and together allows the ontology to work and reason similarly to how humans perceive interlinked concepts.

1.1.2 Knowledge Graphs

A knowledge graph is a knowledge base that uses a graph-structured data model. It is a network of entities, their semantic types, properties, and relationships. One of its main characteristics is that it's generally built with free-form semantics and therefore entities are not necessarily based on classes and they are not inserted respecting a data structure. The structure of the data can, on the other hand, be deducted based on the data which is contained in the graph. It's possible to represent data contained in a knowledge graph as a list of triplets subject - verb - object.
Here are more complete definitions of knowledge graphs: *"A knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge"* [6]. *"A knowledge graph is a multirelational graph composed of entities and relations which are regarded as nodes and different types of edges, respectively"* [7].

1.1.3 Ontologies vs Knowledge Graphs

The main difference which separates the concept of Knowledge graph from the one of Ontology is the focus of what is represented. Describing a domain, Ontologies sharpen the metadata/schema of it, while knowledge graphs concentrate on real data, possibly allowing to find patterns and updating the metadata itself. It's, therefore, possible to say that an ontology, combined with real data, creates a knowledge graph.

1.1.4 Web semantic standards

Web semantic goal is to make Internet data machine-readable. Machines can not easily interact directly with unstructured information and a standard needed to be created. W3C is the entity in charge of defining, maintaining, and improve these standards. To do so, a standard needed to be defined. The main components of this standard are RDF, RDFS, OWL, and SPARQL.

RDF

As described by D. Tomaszuk [8], *“The Resource Description Framework (RDF) [9] is a standard data model proposed by the World Wide Web Consortium (W3C) to describe resources (i.e. real or abstract things) occurring in any application domain. The “description of a resource” means an explicit representation of the attributes and relationship of the resource.”*

In the 1990s, Tim Bray formulated RDF at Netscape as a meta-data schema for describing things. The idea is simple and it’s based on the triple concept reported above. RDF files are therefore composed of a set of logical assertions of the form `subject-predicate-object`. This construct is based on a 3 degrees of freedom format, enabling the description of anything describable. Here comes the real difference from systems based on traditional tables and databases which only had 2 degrees of freedom, where external logic is needed to interpret them.

RDF goes even beyond the idea of a subject-predicate-object triple. It also defines that subject and predicate have to be expressed as a URL, while the object could either be a URL or a literal. This means that by loading the URL of a predicate in a browser, it’s possible to find its definition. The downside of it is quite obvious, as URLs are much longer than literals or variable names, but this was mitigated with the introduction of prefixes. A prefix is a common part of a URL that can be defined before its usage and preponed to the variable name, shortening it. For instance, instead of writing http://example.com/myClass, it’s possible to define once prefix pref = https://example.com/ and use pref:myClass.

In addition to this, RDF introduces an important feature to the basic triple form: a set of reusable predicates. The most important is the predicate rdf:type, which allows to define a hierarchical structure, where entities have a particular type. For example, pref:cat → rdf:type → pref:animal. Using this construct, we can infer things, taking advantage of the fact that, for example, a cat is an animal and not a human.

RDFS and OWL

D. Tomaszuk [8] also describes how OWL and RDFS are extension of RDF and add more reusable predicates to the base standard. *“The Web Ontology Language (OWL) is a W3C recommendation designed to describe ontologies. Specifically, it allows to describe classes, properties, individuals, and data values. In order to describe a domain of interest, OWL defines a set of terms – often called a vocabulary – where each term has a specific meaning, e.g. the term owl:Class represents the class of all classes of resources. An OWL ontology can* *be described using RDF with a precise formal meaning."* Most important OWL terms will follow:

- The term `owl:Class` identifies the class of resources that are RDF classes. An RDF triple (C, `rdf:type`, `owl:Class`) defines that C is a class in the data domain.
- (C_1, `rdfs:subClassOf`, C_2) defines that C_1 is a subclass of C_2.
- `owl:DatatypeProperty` identifies the class of properties that link objects to data values. The triple expression (P, `rdf:type`, `owl:DatatypeProperty`) defines that P is a datatype property.
- `owl:ObjectProperty` identifies the class of properties that relates objects to other objects. The expression (P, `rdf:type`, `owl:ObjectProperty`) defines that P is an object property.
- (P, `rdfs:domain`, C) defines that the resource class C is the domain of the predicate P.
- (P, `rdfs:range`, C) defines that the resource class C is the range of the predicate P.
- (P_1, `rdfs:subPropertyOf`, P_2) defines that P_1 is a sub property of P_2.
- (R, `rdfs:label`, *lab*) defines that *lab* is a human readable label (name) for the resource R.
- (R, `rdfs:comment`, *com*) defines that *com* is a human readable description of the resource R.
- (C, `owl:unionOf`, *list*) defines that C is the union of the classes in the collection *list*.
- (P, `rdf:type`, `owl:AnnotationProperty`) defines that P is an annotation property.
- (R, `rdf:type`, `rdfs:Datatype`) defines that R is a (personalized) datatype.

SPARQL

Since RDF was proposed, the problem of how to query data in this format was raised. As reported by Jorge Perèz [10], in 2004 the RDF Data Access Working Group (part of the Semantic Web Activity) released a first public working draft of a query language for RDF, called SPARQL. Now, SPARQL is the recommended query language for RDF from W3C.

Essentially, it's a Graph-matching query language, and given a data source (possibly multiple sources) it's possible to run a query against it to get results. The query is composed of three parts: pattern matching, solution modifiers, and output. The *pattern matching* enables to choose the data source, filtering values of possible matchings, and specify constraints for the query. The *solution modifiers* allows modifying the output computed applying operators like projection, distinct, order, limit, and offset. The *output* is the actual final result of the query and can be in the form of a boolean, selection of values, construction of new triples, and descriptions of resources, as explained in [10].

1.2 Graph types

Graph databases have become more and more important for several reasons like flexibility and extensibility. It's not always clear though the difference between the two main types of graph technologies: semantic graphs and property graphs. A good understanding of the major difference between the two is written in an article by Jans Aasman [11], that also says:

"For simple graph oriented data relationships, a non-semantic (or property graph) database approach might solve a single dimensional problem like: shortest path, one-to-many relationships, weighted elements, structured inter-relationships.
But rarely are problems and queries that simple. Real world data is highly complex, multi-dimensional, and needs the powerful additional features of a semantic graph solution.
Also, property graphs require the fixed definition of classes/objects, types, and nodes – basically a fixed schema. If the data or data sources ever change, those changes need to be coded before any new data can be accessed."

On the other hand, it's not to be undervalued the complexity of RDF, the standard on which semantic graph models are based on. In the book "Validating RDF Data" by Dan Brickley and Libby Miller they explain their concern about these models:

"People think RDF is a pain because it is complicated. The truth is even worse. RDF is painfully simplistic, but it allows you to work with real-world data and problems that are horribly complicated" [12].

It's hard to state uniquely which technologies should be used, as it mainly depends on the user's need. Summing up, while one model has a data-centric focus, the other has a more knowledge-centric way of representing data.

Figure 1.1 shows the most common graph database engines that exist today, dividing them in two categories (semantics and property graphs), while Figure 1.2 shows which technologies are in use underneath. The following sections will describe the two major variants of graph databases and some of the standards and technologies that have been built on top of them.

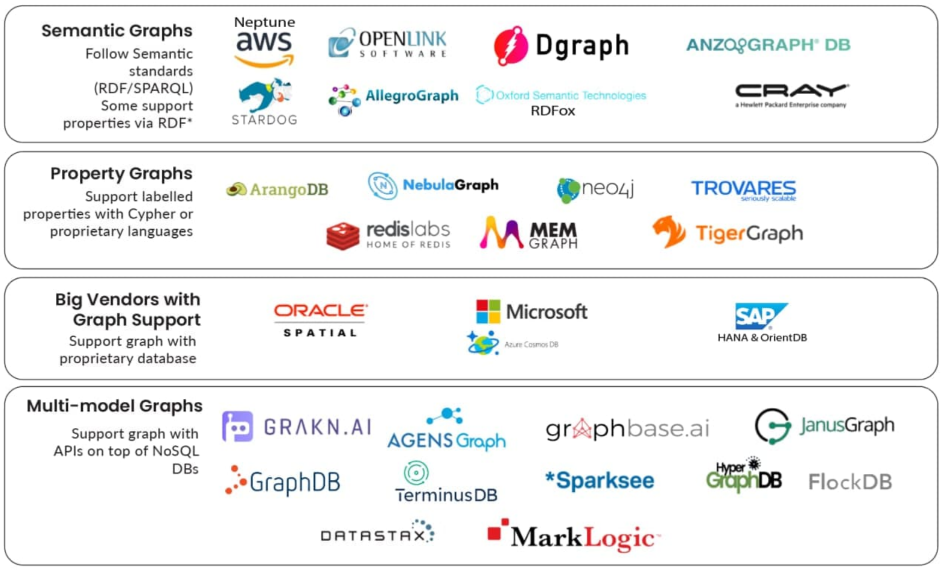

Figure 1.1: Graph Types Model Comparison.

Obtained from https://medium.com/swlh/top-seven-hits-and-misses-in-the-graph-database-world-ab98cc785c61

1.2.1 Semantic Graphs

Semantic graphs are those which follow semantic standards: RDF, SPARQL, and OWL. They can store complex data that represent the real world. It's based on a construct commonly called a "triple", which has three key parts to store concept and context, in the most granular and atomic form. For example, a statement like "Samuele is a son of Massimiliano", three parts are identified: `Samuele` - the Subject, `is a son of` - the Predicate, and `Massimiliano` - the Object. Using semantic graphs, it's also possible to understand relationships that are not explicitly defined, inferring them. In fact, in case another statement is "Massimiliano is a son of Alberto", it's possible to infer that `Samuele` is a grandson of `Alberto`. Also, it would be possible to know that both `Samuele` and `Massimiliano` are sons and that both `Massimiliano` and `Alberto` are fathers.

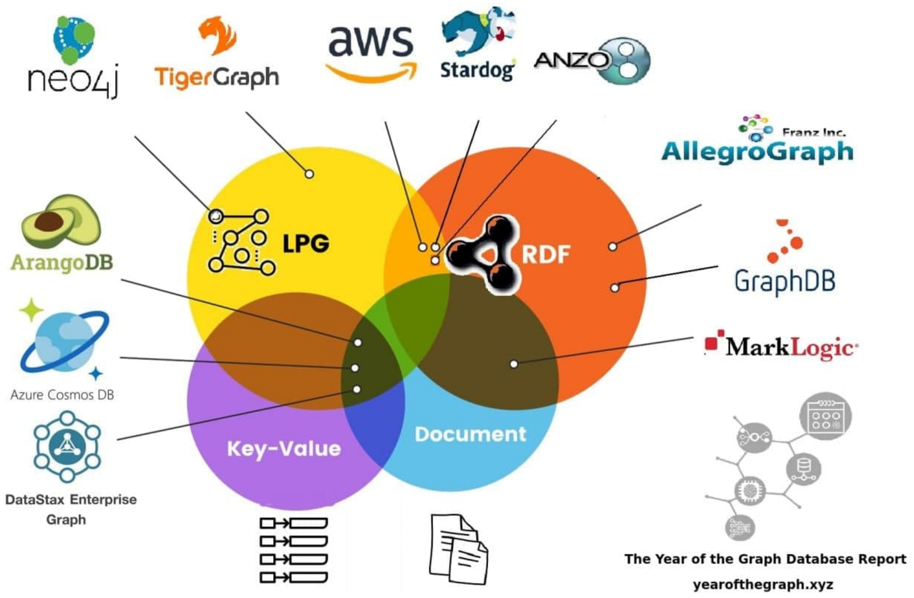

Figure 1.2: Graph Types Technology comparison.

1.2.2 Labeled Property Graphs

Labeled property graphs are the second major variant of graph databases. They're much simpler than RDF and are composed of a set of nodes and a set of edges. Each element in this type of graphs is like a struct with a set of key-value fields, which describe the instance itself. Edges, in this case, are not serialized as part of a triple but are a special property of the node, where the value is just a pointer to another node.

Looking at Figure 1.3 it's visible how the popularity of Neo4j, one of the most common Labelled Property Graph (LPG), is exponentially higher than other types of graph databases.

There are many reasons behind this. One of these is certainly the pragmatic approach with which LPGs have been built, creating a database that ordinary humans could understand. Instead of being focused on semantics like RDF graphs, property graphs were always fundamentally focused on the graph as the best abstraction for modeling the world.

From a low-level point of view, on the contrary, RDF has a pure graph structure, whereas LPGs are not easily implemented straightforwardly. One consequence of this is that the edges of triples cannot have properties natively. Even if there are several ways of obviating this problem, there's no particular interest by semantic web academic community in creating a standard, like to build more complex data-structure in RDF which represent qualified edges.

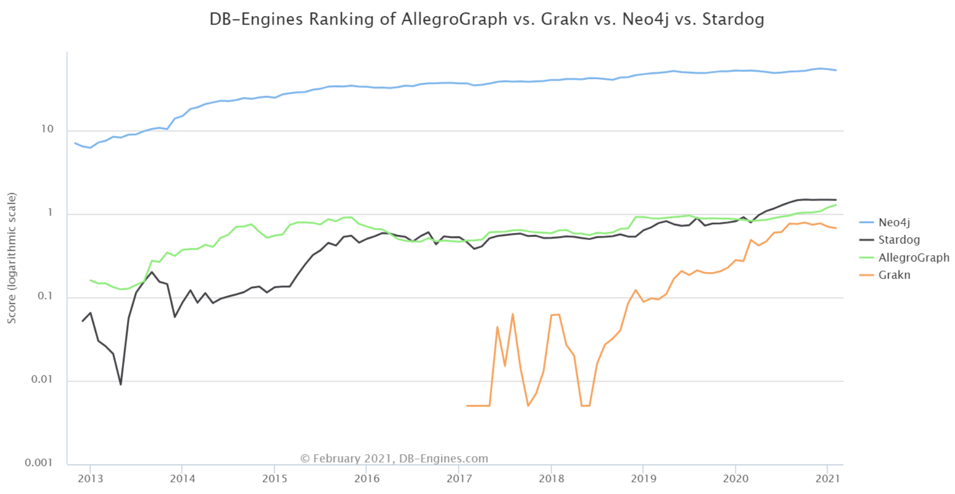

Figure 1.3: DB engines popularity [1]

The tendency is to stay more focused on expressiveness and inference.

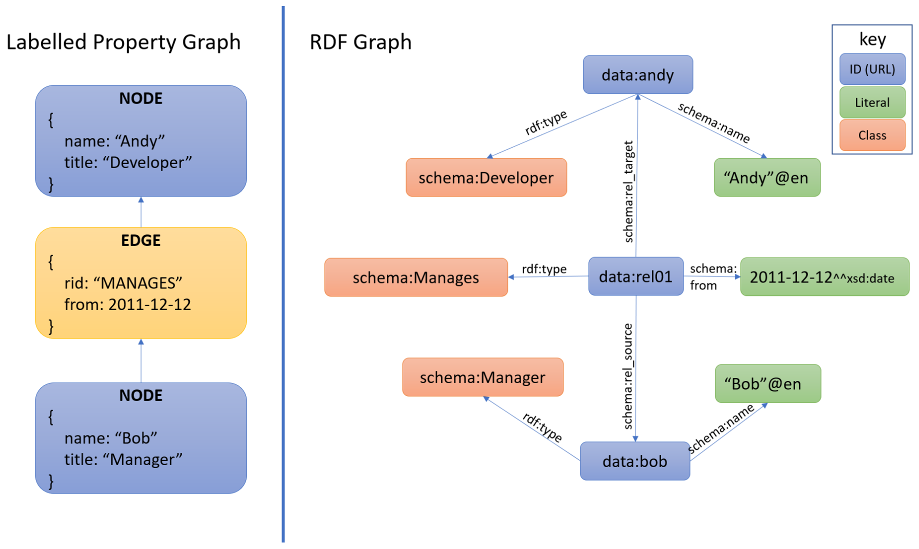

Figure 1.4: RDF vs LPG triple representation

Obtained from `https://medium.com/terminusdb/graph-fundamentals-part-2-labelled-property-graphs-ba9a8edb5dfe`

1.2.3 Multi-model Graphs

As Figure 1.1 suggests, some graph databases use multi-model systems beneath and support graph APIs on top of NoSQL DBs. One of the most known is grakn.ai[1].

1.3 Linked Open Data

Data today is the new oil and for many learning software, the amount of data they can access and use it's critical to its success. It would be ideal for such software to be able to access, use and contribute to all the data available on the web in an interlinked way. It's already possible to access many resources, but as Bauer highlights in its article, two main problems slow down this process:

"First of all, databases are still seen as silos, and people often do not want others to touch the database for which they are responsible. This way of thinking is based on some assumptions from the 1970s: that only a handful of experts can deal with databases and that only the IT department's inner circle can understand the schema and the meaning of the data. This is obsolete. In today's internet age, millions of developers can build valuable applications whenever they get interesting data."

"Secondly, data is still locked up in certain applications. The technical problem with today's most common information architecture is that metadata and schema information are not separated well from application logics. Data cannot be re-used as easily as it should be. If someone designs a database, he or she often knows the certain application to be built on top. If we stop emphasising which applications will use our data and focus instead on a meaningful description of the data itself, we will gain more momentum in the long run. At its core, Open Data means that the data is open to any kind of application and this can be achieved if we use open standards like RDF to describe metadata" [13].

Figure 1.5 shows how the web has quite turned in this direction during the last years though, creating an interlinked knowledge based on the RDF model, each called Linked Open Data (LOD). The so-called LOD Cloud [14] covers now more than 50 billions entities, linking many different facts and areas. The most known ones include DBPedia[2], Freebase[3], Linked GeoData[4] etc. Figure 1.6 shows how many LOD contribute nowadays to build the LOD cloud.

There are many examples of real usage which express how important is that data is available and connected, like Linked Data in biomedicine [15, 16] or in Linked government data [17, 18].

[1]`https://grakn.ai`
[2]`https://wiki.dbpedia.org/`
[3]`http://www.freebase.com/`
[4]`http://linkedgeodata.org/`

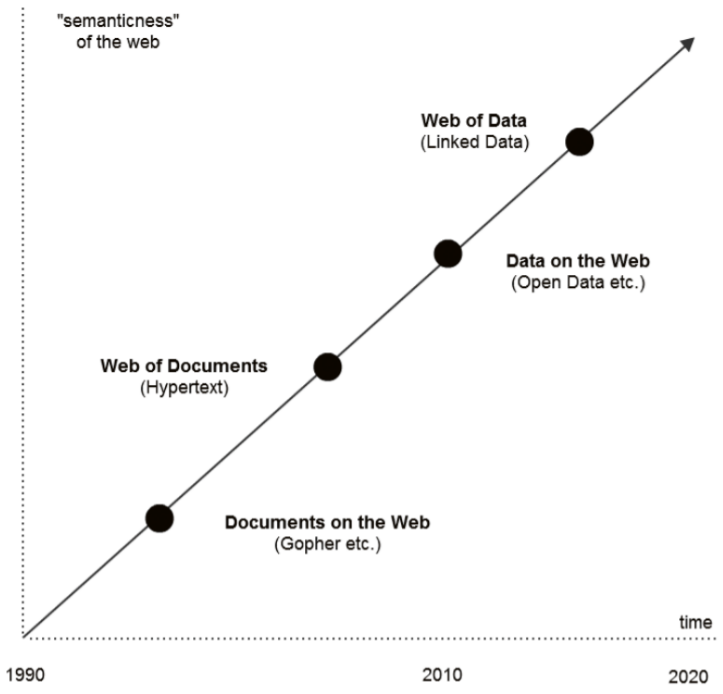
"semanticness" of the web
Web of Data (Linked Data)
Data on the Web (Open Data etc.)
Web of Documents (Hypertext)
Documents on the Web (Gopher etc.)
time
1990
2010
2020

Figure 1.5: Semanticness of the web through years

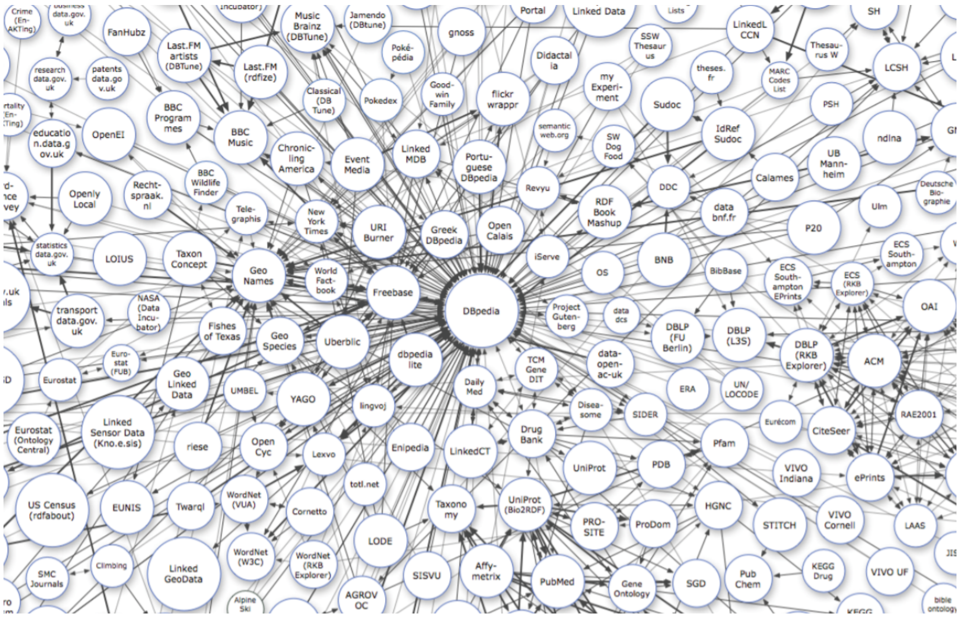

Figure 1.6: LOD cloud

1.4 Time-Evolving Graph

Recently, we note that most of the time, data used to build a knowledge graph gets continuously changed and integrated with more data. Examples of this are KG based on social media, where posts are published every day, or less evidently, on Medicine research, where discoveries update the knowledge in the field. However, most algorithms for KG embedding have been designed for static data, which lead to low efficiency and high error rate. Therefore, modeling dynamically-evolving, multi-relational graph data has received a surge of interest with the rapid growth of heterogeneous event data. And recent research has focused on temporal knowledge graphs and their temporal information [19].

Chapter 2

Context

2.1 The Voice of Patients

A disease, in order to be defined rare, needs to have a prevalence (number of cases over a given population) which doesn't exceed a threshold established by each country. Many people are touched by rare disease challenges, among which there are patients, families, caregivers, clinicians and researchers.

Official entities like *Orphanet*[20] report more than 6000 registered rare diseases (but up to 7000 are estimated) and more than 350 million people live with them every day (5% of the world population). This means that, even if the percentage of lives involved by a single RD is limited by definition, the total number of patients is not limited at all.

Furthermore, almost the totality of rare diseases is not curable and only its 5% has therapeutic options. 80% of these have genetic origin and 50% of people affected by a RDs is referred to children (with a mortality rate of 35% in their first 5 years of living)[21, 22, 23]. Moreover, diagnoses of these pathologies are often belated or wrong for a long period of time (averagely 4.8 years and 7.3 doctors visited, per RDs), leading to a worsening of the disease itself[24, 25].

In this field, information is scarcely available and disintegrated, and in recent years we are witnessing a strong growth of patient communities on social platforms such as *Facebook*. People with these diseases are often feeling lonely in this challenge and consequently in need to dialogue with others in the same conditions. Social contexts become therefore the place where experiences, opinions and information are shared, during the whole path of a rare disease patient (from symptomatology to diagnosis, from therapeutic treatments to specialized centers, from doctors of reference to the impact on lifestyle) [26].

2.2 POIROT

In order to gather information from social media we'll use a technique for semantic correlation extractions between terms and documents which is based on POIROT [2].

In the following sections it'll be explained the operation of this methodology and other useful aspects of my contribution in this thesis.

POIROT will be reported because knowledge will be extracted from text with this methodology and because authors have already applied and tested it (obtaining a F1 measure of 79%) with the same study case considered in this thesis. Its research has been developed by an original contribution of professor Gianluca Moro and it recently won the Best Paper Award.

Furthermore, the way these correlations are reported on the knowledge graph follows the same principles proposed by the authors in KDIR [3].

Authors of the study cited above, argue that a right step towards a future of inclusion can be the extraction of knowledge from text and to find phenomena explanation of various kinds in it. This would be a future where data concerning personal experiences (also called "real world") of patients don't get lost. Solutions of this kind can have a huge impact for caregivers and researches, but it's also an interesting challenge in NLP, considering that posts and comments are short, unlabeled, full of noise, and with a lot of grammatical imperfections [2].

2.2.1 Methodology

As also reported by authors in the Springer extension[2], it's becoming increasingly important to learn knowledge from text as the amount of unstructured content on the Web rapidly grows. Despite recent breakthroughs in natural language understanding, the explanation of phenomena from textual documents is still a difficult and poorly addressed problem. Additionally, current NLP solutions often require labeled data, are domain-dependent, and based on black box models. In this thesis we're using POIROT [2], which is a descriptive text mining methodology for phenomena explanation. It has been designed to provide accurate and interpretable results in unsupervised settings, and quantifying them based on their statistical significance.

Discovering the reasons that explain some phenomena expressed within a corpus of textual documents requires bringing out semantic relationships among unbounded combinations of relevant concepts, such as symptoms, treatments, drugs, and foods.

The solution in use is domain and language independent and consists of various modules whose implementation can be adapted to the specific problem under consideration, as shown in Figure 2.1: quality preprocessing (to improve the quality of the text documents contained within the starting corpus), document classification (to recognize the phenomenon to be investigated), analysis preprocessing (to prepare the data for the analysis), term weighting (to identify the significance of each term in each document, defining also the vocabulary), and language modeling (to bring out semantic similarities between terms and documents within a low-dimensional latent vector space).

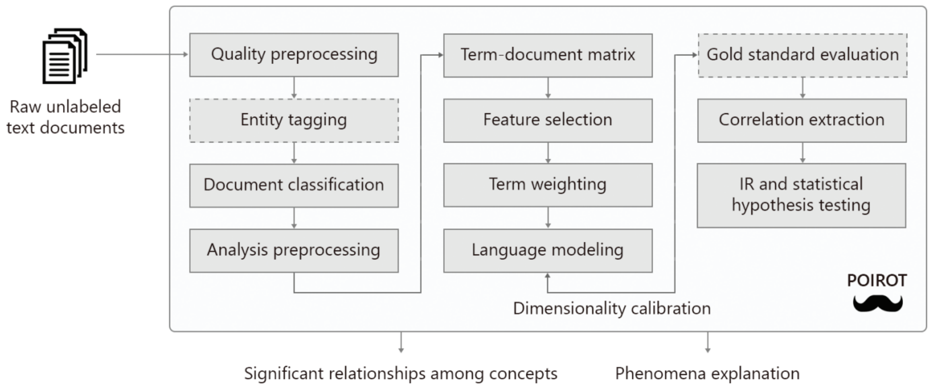

Figure 2.1: Poirot modularity

Information retrieval and statistical hypothesis testing in the space just constructed makes it possible to derive and quantify semantic correlations between terms, documents, and classes. The generation of a textual explanation is incrementally carried out on a query, which is like an artificial document.

As also explained by authors in their publications [3, 27], after identifying a first term or considering a starting query, at each step the query is folded into latent space. From here, it searches for new terms semantically close to the query and with greater significance, choosing the one that — if combined with the current description — continues to be representative of the phenomenon. The degree of correlation between the query and the phenomenon is indicated by the p-value resulting from the application of the chi-squared (χ^2) statistical hypothesis test, together with R-precision. The process ends when it is no longer possible to enrich the query and remain below the pre-established p-value threshold.

Precisely, POIROT projects relevant terms and documents in a latent semantic space and statistically reports significant correlations between them. Next, those areas linked to the phenomenon of interest which have an unexpected concentration are investigated, and it constructs interpretable global textual explanations by iterative refining, keeping trace of accurate probabilistic information.

Authors do this in several ways, in DATA [27] they do it with Latent Semantic Analysis (LSA) [28, 29] and in POIROT they also experimented a probabilistic language model like Latent Dirichlet Allocation (LDA) [30] and Probabilistic Latent Semantic Analysis (pLSA) [31].

LSA fold-in and incremental query enrichment

LSA is a powerful technique for capturing knowledge that closely matches human semantic similarities. It is based on Singular Value Decomposition (SVD) [32] applied to text, and is based on the decomposition of the matrix terms-documents into the matrices U, Σ and V^T, as shown in Figure 2.2.

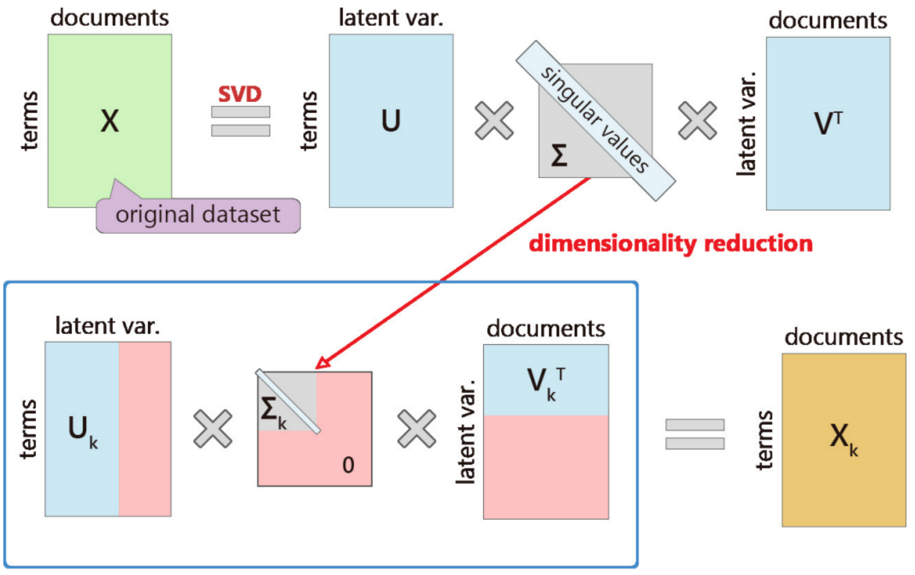

Figure 2.2: LSA matrix decomposition and their reduction through SVD [2]

We only get into details of LSA because it is the technique which lead the author to reach their best results and consequently the reference implementation for this thesis.

As also the author report in POIROT presentation [2], LSA performs a mapping of a term-document matrix (eventually weighted) into a low-dimensional latent semantic space. The mapping is based on Singular Value Decomposition (SVD), a linear algebra technique that factorizes a matrix C into the product of three separate matrices.

U and V (Figure 2.2) are two orthogonal matrices, and Σ is a diagonal matrix containing the singular values of C in descending order. The singular values in Σ are the components of the new dimensions, and the first of them captures the greatest variation of the data (i.e., contain more information). SVD reduces dimensionality by selecting only the largest k singular values, and only keeping the first k columns of U, and the first k rows of V^T. The positions of all terms and documents in the latent semantic space are obtained from the products of matrices $U_k \times \Sigma_k$ and $V_k \times \Sigma_k$, respectively. The value of the hyperparameter k can be calibrated to delete noise and unnecessary data, as well as better capture the mutual implications of terms.

The incremental process for the construction of explanations for a certain phenomenon of interest — representing the last part of POIROT, based on techniques of information retrieval and test of statich hypothesis — adopts the fold-in technique. The fold-in consists in the determination of an embedding of a new document in the space without requiring its regeneration, realizing the transposition of queries in the latent semantic space. Clearly, those query must undergo the same preliminary transformations that the cell entries of the matrix received before model construction (i.e., quality preprocessing, entity tagging, analysis preprocessing, term weighting).

2.2.2 Application to Knowledge Graph Learning

In a recent paper [3], author also showed a possible application of POIROT to the task in Knowledge graph learning in acquiring knowledge from a collection of short, unstructured and unlabeled texts, through a custom implementation of learning layer cake. In the paper they show how to do an extraction of relational facts from plain text, keeping interpretability and the interrogability of the results. It is actually one of the main approaches for the construction and expansion of KGs.

The subject which studies the mechanism to transform the creation and updating of ontologies into a semi or completely automatic process is Ontology learning (OL).

The process of ontology acquisition directly from unstructured text is been recently subject of attempts of automation as a result of the huge increase in magnitude and throughput related to the generation of textual content. Consequently, the development of NLP and advanced machine learning approaches — essential to extract knowledge from text document — goes simultaneously with the development of OL.

OL and Knowledge graph learning (KGL) from text follow the same principles, usually based on a multistep approach known as learning layer cake (Figure 2.3). It's normally based on the following steps: the extraction of terms and their synonyms from the underlying text, the formation of concepts through the combination combination of them, the identification of taxonomic and non-taxonomic relationships between the found concepts, and finally the generation of rules.

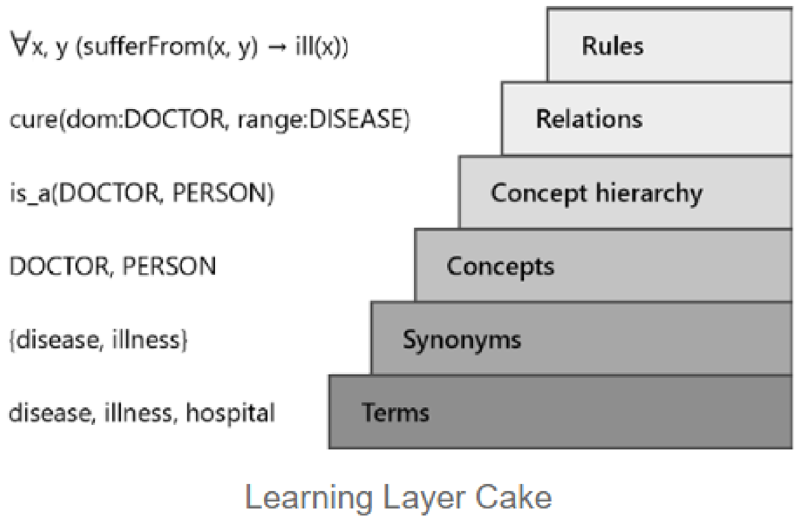

Figure 2.3: Learning layer cake [3]

In their last paper [3], POIROT authors also demonstrated how a POIROT extension can be effectively applied to this task and how KGs enhance the expressive power, interrogability, and interpretability of the extracted knowledge.

Compared to POIROT, the output of this new module is a KG instead of a flat set of clusters made up of correlated unlabeled terms. The graph also includes some terms that are recognized as entities within hierarchical taxonomy and therefore correlations are interconnected and not independent from each other. This means an enhancement in expressiveness.

Named Entity Recognition (NER) [33] system introduced a typing of terms is an important step towards interpretability, allowing users which are not expert of the domain to better understand what a certain term represents (e.g., canary is_a /animal/bird).

This greater expressive power also leads to new forms of interrogability. For instance, as also the authors report [3]: *"if a user wants to investigate all the significant correlations between two or more types of entities (e.g., drug ↔ symptom, symptom ↔ food), he is no longer forced to check them individually and to know all the terms related to the instances of the types considered (e.g., <"lansoprazole gerd": ?>, <"aspirin headache": ?>, . . .). Now it is possible to manage queries concerning the meta-levels of the concept hierarchy."*

On the obtained graph, authors also showed advantages resulted from what reported through the execution of several Semantic Query-enhanced Web Rule Language (SQWRL).

SQWRL queries were executed on the KG learned from the textual corpus and their results show the potential deriving from the new meta-level knowledge introduced in the system.

2.2.3 Dataset

The application is built upon data regarding the Esophageal Achalasia (ORPHA:930), a rare disorder of the esophagus characterized by the inability of the lower esophageal sphincter (LES) to relax. Collaborating with *Associazione Malati Acalasia Esofagea (AMAE)*[1][2] — the main Italian patient organization for the disease under consideration — we keep downloading anonymous text documents from the Facebook Group directly managed from it[3] (with ≈ 2000 current users and > 10 years of history). We're using the same dataset that were used in previous work about POIROT and its application in the world of graph learning [27, 26, 2, 3] As of 5/8/2019 the dataset consisted of 6,917 posts and 61,692 first-level comments, published between 21/02/2009 and 05/08/2019, and it is increasing every day with each new post.

2.2.4 Limitations

The POIROT model is definitely flexible, modular and an expressive method to convert unstructured data to usable information. It also is independent from any neural model language, which, in NLP, is often representing a black-box caused from the high number of dimensions of those models, like T5 [34] or GPT-3 [35]. Nevertheless, in its actual form, it presents some limits which prevent to unlock its potential.

- First of all it has to be run in batches. Poirot algorithm needs to take into account the whole text in order to understand the context and create weighted relationship between terms. The world, on the contrary, is dynamic and changes quickly.

  In POIROT, authors concentrated on text from a social network which was gathered in batch, but they haven't focused yet on integrating knowledge based only on the text of new posts. It would be quite impractical to run the algorithm on the new whole data set at every post. An incremental approach would be more practical instead, taking advantage of previously calculated connections.

[1]`https://www.amae.it/`

[2]`https://www.orpha.net/consor/cgi-bin/SupportGroup_Search.php?lng=EN&data_id=106412`

[3]`https://www.facebook.com/groups/36705181245/`

- The knowledge graph generated is static and it's not evolving over time. It's also just stored in an OWL file and needs an intermediary software to be queried or analyzed.
- Temporal analysis and changes over time can not be recorded and analyzed.
- In the example case, text is taken from a single source only. Even if the *Facebook* group seems to be quite active so far, people are changing their favorite social media quite rapidly and enabling multiple sources would grant more accuracy and completeness of information.

2.3 Goals

As primarily objective of the proposed project is to capture the "voice of patients" from their messages on social media communities through the usage of the POIROT methodology.

We also want to overcome the limitations cited before building a general system which periodically queries (possibly many) data sources, and uses the new text to incrementally update the knowledge graph specifically for the considered Rare Disease: Esophageal Achalasia. This RD is taken as an example to build a knowledge graph upon it, but the system should be applicable to any type of text sources from which learning would be meaningful. It is therefore important to keep it general and topic-agnostic, with interchangeable components.

Chapter 3

Knowledge Graph tools

3.1 Overview

As previously seen, there are many tools to interact with data models. The main differences between them regard the primary database model (RDF vs LPG vs Multi-model), the subject of study (KG or Ontologies), the presence of a reasoner and the available methods to access data.

3.1.1 Protégé

Protégé is, from its website[1], *a free, open-source platform that provides a growing user community with a suite of tools to construct domain models and knowledge-based applications with ontologies.* It's developed by Stanford University and is called "the leading ontological engineering tool", with more than 360,000 registered users (as per their website). It's written in Java and is used to edit ontologies and manage knowledge. Provides a graphic user interface to define ontologies. It also has reasoning capabilities thanks to its deductive classifier, used to validate model consistency and infer implicit information analysing the ontology itself. Two types of UI are available: the WebUI, WebProtégé, and the downloadable desktop version, Protégé Desktop.

Protégé is supported by a strong community of academic, government, and corporate users, who use Protégé to build knowledge-based solutions in areas as diverse as biomedicine, e-commerce, and organizational modeling.

It stores data on RDF files and is not a database engine.

[1]`https://protege.stanford.edu/products.php`

3.1.2 Allegrograph

Allegrograph is a private software owned by Franz Inc., designed to store RDF triples meeting W3C standards. It can also store document-oriented informations, in JSON-LD format. This differentiates it from other property graph databases, since it also allows document storage with contextual and conceptual intelligence.

It is in use in commmercial projects, a US Deparment of Defense project, and in the TwitLogic project, that is bringing Semantic Web to Twitter data. Its functionalities are accessible using programming languages like Java, Python, Commond Lips and other APIs.

AllegroGraph is W3C/ISO standards compliant and supports JSON, JSON-LD, SPARQL 1.1, OWL Reasoning, SHACL, and Prolog rules and reasoning directly and from numerous client applications.

3.1.3 Grakn

Grakn is an open-source, distributed knowledge graph database for knowledge-oriented systems. It is evolved from the relational database for highly inter-connected data because it provides a concept-level schema that implements the Entity-Relationship model completely. It implements the principles of knowledge representation and reasoning. Graql is Grakn's declerative query language which enables deductive reasoning and analytic query language over large amount of complex data. Effectively it's remarked as a good base for AI and cognitive computing systems. From their website[2], they boast an intuitive and expressive knowledge schema, an intelligent automated reasoning in real time, distributed analytics and an Higher-level language.

3.1.4 Neo4j

Neo4j is a native graph database, built from the ground up to leverage not only data but also data relationships. Neo4j connects data as it's stored, enabling queries never before imagined, at speeds never thought possible, as they report on they website[3]. It's an open source graph database management system developed in Java by Neo4j, Inc. It's also ACID compliant with native graph storage and processing. Even if it's written in Java, it exposes HTTP endpoint accessible using the Cypher query language.

Cypher is a graph-optimized language that understands stored connections in Neo4J, since all connections are stored and not computed at query time.

[2]`https://grakn.ai`

[3]`https://neo4j.com`

Data are not stored in tables and therefore, differently from SQL, JOINS are not present in the query language, making syntax smoother and shorter. A comparison example is reported from their website.

Cypher:

```
MATCH (p:Product)-[:CATEGORY]->
    (l:ProductCategory)-[:PARENT*0..]->
    (:ProductCategory {name:"Dairy Products"})
RETURN p.name
```

SQL:

```
SELECT p.ProductName
FROM Product AS p
JOIN ProductCategory pc
    ON (p.CategoryID = pc.CategoryID AND
        pc.CategoryName = "Dairy Products")

JOIN ProductCategory pc1
    ON (p.CategoryID = pc1.CategoryID)
JOIN ProductCategory pc2
    ON (pc1.ParentID = pc2.CategoryID
        AND pc2.CategoryName = "Dairy Products")

JOIN ProductCategory pc3
    ON (p.CategoryID = pc3.CategoryID)
JOIN ProductCategory pc4
    ON (pc3.ParentID = pc4.CategoryID)
JOIN ProductCategory pc5
    ON (pc4.ParentID = pc5.CategoryID
        AND pc5.CategoryName = "Dairy Products");
```

3.1.5 Stardog

Stardog is a commercial RDF database with SPARQL query, transactions, and world-class OWL reasoning support. The product is available to be downloaded or used online on Stardog Cloud. A 30 days trial is available for free to everyone, but a 1-year license also is available for academic purposes.

Stardog also provides Stardog Studio, the GUI to browse connections in data, visualize results to see and customize the knowledge graph. Stardog Studio is based on Electron, an open-source Javascript framework used to write desktop application in Javascript. Being based on Javascript it also allows the program to be run on the browser: Stardog Studio Web UI[4].

To interact with Stardog databases there are several way. The first one is the command line. It's easy as

```
stardog query execute --reasoning http://myHost:9090/myDb
"select * where { ?s ?p ?o }"
```

to execute a query directly to the database. It also provides libraries to be accessed by programming languages like .Net, Clojure, Groovy, Java, JavaScript, Python, and Ruby, other than being accessible via HTTP API.

Another key factor of Stardog is the detailed documentation available at: `https://docs.stardog.com`.

3.2 Comparison with table

Among the many tools that are available online to manage Ontologies and Knowledge graphs, Table 3.1 compares the main ones taking many parameters into account. The table has been created starting from the one generated on by db-engines.com [36] and later integrated with other parameters of interest, like the storage system, possibility to query the database, reasoner availability, visualization tool, and the presence of an IDE.

[4]`https://stardog.studio`

Table 3.1: Db engines compared using DB-engines.com and subsequently integrated.

Begin of Table					
	Protégé	**AllegroGraph**	**Grakn**	**Neo4J**	**Stardog**
Description	A free, open-source ontology editor and framework for building intelligent systems	High performance, persistent RDF store with additional support for Graph DBMS	Grakn is a distributed, hyper-relational database for managing complex data that serves as a knowledge base for cognitive/AI systems.	Open source graph database	Enterprise Knowledge Graph platform and graph DBMS with high availability, high-performance reasoning, and virtualization
Primary database model	RDF/OWL file	Document store info Graph DBMS RDF store	Graph DBMS Relational DBMS info	Graph DBMS	Graph DBMS RDF store
DB-engines ranking	na	Score 1.19 #156 Overall #26 Document stores #13 Graph DBMS #7 RDF stores	Score 0.70 #203 Overall #18 Graph DBMS #100 Relational DBMS	Score 53.79 #19 Overall #1 Graph DBMS	Score 1.47 #144 Overall #11 Graph DBMS #6 RDF stores

Continuation of Table 3.1

	Protégé	**AllegroGraph**	**Grakn**	**Neo4J**	**Stardog**
Website	protege .stanford.edu	allegrograph.com	grakn.ai	neo4j.com	stardog.com
Technical documentation	protegeproject .github.io /protege/	franz.com/ agraph/support/ documentation/ current/	dev.grakn.ai/docs	neo4j.com/docs	stardog.com/docs
Developer	Stanford Center for Biomedical Informatics Research	Franz Inc.	Grakn Labs	Neo4J Inc.	Stardog-Union
Initial release	1999	2004	2016	2007	2010
Current release	5.5.0, Mar 2019	7.0.0, Apr 2020	1.8.4, Nov 2020	4.2.2, Jan 2021	7.3.0, May 2020
License	Open Source	commercial, limited free	Open Source	Open Source	commercial
Data scheme	yes	yes	yes	schema-free and schema-optional	schema-free and OWL/RDFS-schema support

Continuation of Table 3.1

	Protégé	**AllegroGraph**	**Grakn**	**Neo4J**	**Stardog**
SQL	SPARQL	SPARQL is used as query language	no	no	Yes, compatible with all major SQL variants through dedicated BI/SQL Server
APIs and other access methods	Java library	RESTful HTTP API, SPARQL	console (shell), gRPC protocol, Workbase (visualisation software)	Bolt protoco, Cypher query language, Java API, Neo4j-OGM info, RESTful HTTP API, Spring Data Neo4j, TinkerPop 3	GraphQL query language, HTTP API, Jena RDF API, OWL, RDF4J API, Sesame REST HTTP Protocol, SNARL, SPARQL, Spring Data, Stardog Studio, TinkerPop 3
Clients	Protégé client	C#, Clojure, Java, Lisp, Perl, Python, Ruby, Scala	All JVM based languages, Groovy, Java, JavaScript (Node.js), Python, Scala	.Net, Clojure, Elixir, Go, Groovy, Haskell, Java, JavaScript, Perl, PHP, Python, Ruby, Scala	.Net, Clojure, Groovy, Java, JavaScript, Python, Ruby

Continuation of Table 3.1

	Protégé	**AllegroGraph**	**Grakn**	**Neo4J**	**Stardog**
Server-side scripts	yes	yes	no	yes	yes
Triggers	no	yes	no	yes	yes
Partitioning methods	na	with Federation	Sharding	none	none
Replication methods	na	Multi-source replication, Source-replica replication	Multi-source replication	Causal Clustering using Raft protocol	Multi-source replication in HA-Cluster
MapReduce	no	no	yes	no	no
Consistency concepts	On demand consistency check	Immediate Consistency or Eventual Consistency depending on the configuration	Immediate Consistency	Causal and Eventual Consistency configurable in Causal Cluster setup Immediate Consistency in stand-alone mode	Immediate Consistency in HA-Cluster
Foreign keys	no	no	no	yes	yes
Transaction concepts	no	ACID	ACID	ACID	ACID

Continuation of Table 3.1

	Protégé	**AllegroGraph**	**Grakn**	**Neo4J**	**Stardog**
Concurrency	no	yes	yes	yes	yes
Durability	yes	yes	yes	yes	yes
In memory capabilities	no	no	no	no	yes
User concepts	yes	Users with fine-grained authorization concept, user roles, and pluggable authentication	yes	Users, roles and permissions. Pluggable authentication with supported standards (LDAP, Active Directory, Kerberos)	Access rights for users and roles
Storage	OWL files	Internal	Internal	Internal	Internal
Querying	yes	yes	yes	yes	yes
Reasoning	yes	yes	yes, internally	yes	yes
IDE	yes	yes	yes	yes	yes
Visualization	yes	yes, with external tools	yes	yes	yes
Serialization format	OWL, RDF, Turtle, OBO	na	GQL	proprietary format .db	na

Continuation of Table 3.1

	Protégé	**AllegroGraph**	**Grakn**	**Neo4J**	**Stardog**
Hypergraph	no	no	yes	no	no
Relevant customers	over 360K users	Ford Stanford Deloitte U.S.ARMY ESA NASA Pfizer SIEMENS IBM	na	Allianz Microsoft Lyft UBS Airbnb IBM Financial Times HP Ebay	Nasa U.S. Air Force Bayer BOSCH Ebay Cisco Nokia Siemens Bank of America

3.3 Knowledge graph database vs graph database

Knowledge graphs are graphs, therefore every tool which allows a representation of a graph would be able to maintain the data structured. The main difference between a graph database and a knowledge graph storage is based on the operations for which a knowledge graph is useful. Two of the main operations are querying and reasoning. Querying allows interrogating the graph to extrapolate information from it. The reasoning is a technique that finds implicit connections between nodes and uses them to solve queries enhancing results.

3.4 Chosen tool

In the last part of the project, triples are generated and need to be stored in a graph database. To choose a graph database, there are several aspect to be considered. First of all, the need to be usable from a programming language, so that it can be updated automatically based on time or on events. The second fundamental feature needed is the reasoner. It enables more intelligent queries inferring implicit facts. An other feature which is essential is a schema RDF based, necessary to integrate data with LOD Cloud.

Further important features are a good documentation, to speed-up the development of the application, scalability, in order to extend the product to other contexts or diseases, an IDE to visualize the graph, and possibility to run multiple queries concurrently.

It'd also good that the tool is already in use by well known companies in relevant projects, as guarantee of usability.

It's notable as Stardog, the tool chosen for this project, has all these characteristics.

Chapter 4

Project

The project consists of creating a system that increments the knowledge of a graph built from unstructured data as new text is available. It's a general purpose software, which would work with any text, but specifically tested on the case of a rare disease,Esophageal Achalasia.

The data source is an Italian Facebook group where members have the opportunity to chat about their disease, the symptoms, the cures they undergo, and important information that can be useful to everybody in the group. Using a technique based on Giacomo Frisoni's paper [27], unstructured data is transformed into triples which can be used to update the graph.

Summing up, the system is built as a web service written in Typescript, which periodically queries Facebook APIs to obtain new text, creates new graph triples using a script written in R based on Poirot fold-in technique, and creates a query to store them in the graph database.

The architecture of the graph, as visible in Figure 4.1, consists of a web application running on a Linux box. It doesn't expose any endpoint to the outside so far, since it's based on a polling system. When the server starts, it checks that the initial graph has already been built with the POIROT methodology, and if not, it proceeds running the external code proposed in literature to create it, considering the whole text available in the text source.

Since then, the application periodically queries the text provider in order to look for more available text. In order to decide the best time interval between checks, a trade-off has been made between the willing to keep information updated in real-time, the frequency of post publishing on the *Facebook* group in use, and trying to avoid checks with empty result, given the monthly query limit imposed to developers from *Facebook*.

Once new posts are retrieved from the text provider, the Fold-In script (described in Chapter 2), is used in order to create new triples and to update the graph. Here, it's also important to mention that given a triple, each term of the triple are scanned in order to check whether they're recognized terms, and in that case they get referenced. After this important procedure which prevents to loose information and classifications, the actual query to insert new triples in the graph is generated, following SPARQL standards.

Finally, the application connects to the database (at the moment in the same Linux box, but potentially elsewhere) and runs the queries against it, logging a report of the triples which have been successfully inserted.

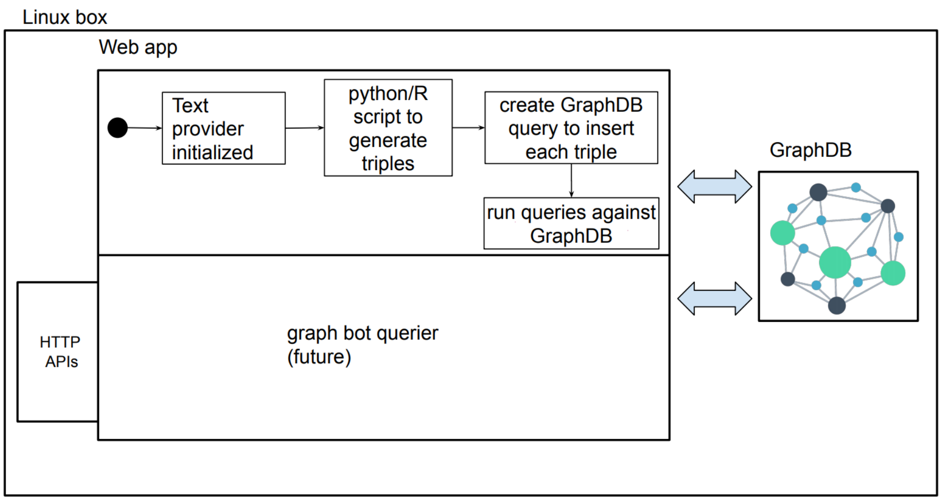
Linux box
Web app
Text provider initialized
python/R script to generate triples
create GraphDB query to insert each triple
run queries against GraphDB
GraphDB
HTTP APIs
graph bot querier (future)

Figure 4.1: Application architecture

4.1 Web server

The web server is written in Typescript, using Node as interpreter. Node is cross-platform and it's one of the most common interpreter used to create full-stack applications. Typescript is the typed JavaScript language, which makes it extremely more consistent and prevents many errors at compile time. Compared to alternatives, this setup also has a huge amount of available libraries and it's easily expandable in the future.

The main class of the application is `Server` and is constituted as follows:

```
class Server {
  private pollingStartingDate: Date;

  private textProvider: TextProvider;

  private graphTriplesGenerator: GraphTriplesGenerator;

  private graphPersister: GraphPersister;
}
```

The `TextProvider` interface provides a method to get the text from whatever source it is.

```
interface TextProvider {
  provideText(): Promise<Array<TextUnit>>;
}
```

The `GraphTriplesGenerator` interface provides a method to convert text to an array of Graph Triples.

```
interface GraphTriplesGenerator {
  generateTriples(textUnit: TextUnit): Array<GraphTriple>;
}
```

The `GraphPersister` interface provides a method to get the text from whatever source it is.

```
interface GraphPersister {
  persistTriples(triples: Array<GraphTriple>): void;
}
```

The main class is responsible for being the main controller of the whole server. It consists of a polling system which check for updates from the Facebook APIs periodically and, if any update is present, the rest of the code is executed. This, translated in Typescript looks as follow:

```
let timerId;

public async startServer() {
    timerId = setInterval(doPollingUnit, MS_ONE_DAY);
}

public stopServer() {
    clearInterval(timerId);
}

private doPollingUnit() {
    this.textProvider
        .provideText()
        .then((textUnits) =>
            textUnits.flatMap((textUnit) =>
                this.graphTriplesGenerator
                    .generateTriples(textUnit),
            ),
        )
        .then((triples) =>
            this.graphPersister.persistTriples(triples));
}
```

4.2 Data source

Facebook provides REST APIs [1] for developers that want to interact with its content. The group which is used is private and therefore admin rights are needed to access data.

A very useful tool which can be used to test the graph APIs is the Graph API Explorer [2]. The application uses the `/feed` endpoint, where it's possible to query for the feed of a specific group or page. A set of parameters can be specified:

- `fields`
  - `id` the id of the element (message, reply, ...)
  - `message` the content of the element (usually text)
  - `comments` possible comments to the element if it's a post or a comment
  - `created_time` the time in which the element was created
  - `updated_time` the time in which the element was updated
  - `picture` link to the picture (in case the element is a picture)
  - `shares` the number of shares of the post
- `since` the lower bound for updated_time parameter
- `until` the upper bound for updated_time parameter
- `limit` the max number of returned element

The application works with most recent data, periodically calling the API in order to get the content of post or comments. A `last_queried_time` variable is stored in the application and updated at every query iteration, using its value in the `since` parameter. An example query can therefore be:

```
/feed?limit=${MAX_LIMIT}&since=${last_queried_time}&fields=id,
created_time,message,picture,shares,comments
```

In order to run the query in Typescript there are two main ways: using the Javascript SDK library[3] or using HTTP requests. The latter seemed to work smoother and be more customizable, therefore a few utilities classes have

[1] `https://developers.facebook.com/docs/graph-api/`
[2] `https://developers.facebook.com/tools/explorer`
[3] `https://developers.facebook.com/docs/javascript`

been written in replacement. Another note on this, is definitely that responses are paged in order to avoid huge payloads or responses and therefore multiple queries are executed automatically per search.

The `FacebookTextProvider` specializes the `TextProvider` and is the data source for this application. In its method `provideText` it fetches all the posts and comments, then extrapolate the text which is contained in them and then returns an array of textUnit, containing that text.

```
export default class FacebookTextProvider implements TextProvider {
  // eslint-disable-next-line no-useless-constructor,no-empty-function
  private lastUpdatedCheck: Date;

  private groupId: string = config.env.groupId;

  private token: string = config.env.token;

  constructor(firstDateToCheck: Date) {
    this.lastUpdatedCheck = firstDateToCheck;
  }

  public async provideText(): Promise<Array<TextUnit>> {
    const solution: Array<TextUnit> = [];
    await fetchFacebookPagePosts({
      pageId: this.groupId,
      fromDate: this.lastUpdatedCheck,
      token: this.token,
      withComments: true,
      withCommentsReplies: true,
    })
      .then((posts) =>
        posts?.forEach((post) =>
          this.addFacebookPostMessageToTexts(post, solution),
        ),
      )
      .catch((reason) => logger.info(`ERROR: ${reason}`));
    this.lastUpdatedCheck = new Date(Date.now());
    return solution;
  }

  private addFacebookCommentMessageToTexts(
    facebookPostComment: FacebookPostComment,
    texts: Array<TextUnit>,
  ) {
    texts.push({ text: facebookPostComment.message });
    if (facebookPostComment.replies !== undefined) {
      facebookPostComment.replies.forEach((reply) =>
        this.addFacebookCommentMessageToTexts(reply, texts),
      );
    }
  }

  private addFacebookPostMessageToTexts(
    facebookPost: FacebookPost,
    texts: Array<TextUnit>,
  ): void {
    if (facebookPost.message !== undefined) {
      texts.push({ text: facebookPost.message });
    }
    if (facebookPost.comments !== undefined) {
      facebookPost.comments.forEach((comment) =>
        this.addFacebookCommentMessageToTexts(comment, texts),
      );
    }
  }
}
```

4.3 Text to graph triples

The triple generation code which is available is written in R and the interaction between the two programming languages is not straightforward. Nevertheless, a library called r-script[4], available as Node module, provides an interface which enables the serialization and the sharing of objects from a Javascript program to the R script and the other way round.

Using this module, it's enough to pass all the previously downloaded text to the R script and get back the graph triples in return.

```
    const scriptFileName = 'foldInScript.R';

    class FoldInGraphTriplesGenerator implements
        GraphTriplesGenerator {
        generateTriples(textUnit: TextUnit): Array<GraphTriple> {
        const out = R(scriptFileName).data(textUnit.text).callSync();
        return JSON.parse(out);
    }
}
```

In case the program is run for the first time or by choice of the user, it's possible to call a particular R script which, using Poirot technique, creates the graph starting from all the posts ever available. Otherwise, if the graph is already present, it's possible to fold-in and update the knowledge creating new triples in a faster way.

We'll adopt the fold-in technique proposed in POIROT and described at Section 2.2.1, embedding new documents in the semantic space previously built, enabling to find the embeddings which are semantically closer associated to both terms and documents, in cosine similarity. The top-n documents and terms above a certain threshold of similarity gets mapped in new triplets (e.g. new_doc → term1, new_doc → term2, and so on). Its implementation can be found at Code 4.1, 4.2 and 4.3.

[4]https://www.npmjs.com/package/r-script

```
new_doc <- makeUserQuery(
  "Gemelli roma",
  userQueryPreprocessingNer,
  applyTermWeightingToDocument,
  tdmw,
  lsa_data$lsam)

# Top-N documents semantically related to the user query
n_nearest <- 10
nearest_docs <- findTopNearestDocuments(
  texts = documents$text,
  dls = lsa_data$dls,
  lsa_dims = lsa_dims,
  query_ls = new_doc$ls_q,
  n = n_nearest)
return(nearest_docs)
```

Code 4.1: Example of makeUserQuery function usage and creation of triplets.

```r
#' Fold-in a custom textual document in a latent semantic space.
#' Apply the same transformation strategy applied to original
    documents.
#' unlike a query, the document is not necessarily composed only of
    the dictionary terms,
#' but is a text freely expressed by a user (similar to a new post).
#'
#' @param doc The custom textual document
#' @param preprocessingDocFun The preprocessing function to apply to
    the document before the fold-in
#' @param termWeightingDocFun The term weighting function to apply on
    the binary query
#' @param tdm The term document matrix to enrich
#' @param lsam The latent semantic space on which perform document
    fold-in
#' @return The document, its binary vector representation, its
    weighted vector, its position in the latent space,
#' its normalized position for visualization purposes, the component
    for similarity calculation
makeUserQuery <- function(doc, preprocessingDocFun,
    termWeightingDocFun, tdm, lsam) {

  # Performs document-level preprocessing
  doc <- preprocessingDocFun(doc)

  # Custom document fold-in
  makeQuery(doc, tdm, termWeightingDocFun, lsam)

}
```

Code 4.2: Function makeUserQuery implementation.

```
#' Perform all the preliminary operations for the calculation of the
    similarity
#' between a query and the terms in the latent space
#'
#' @param q The query (a set of key words to research in the
    documents)
#' @param tdm The term document matrix
#' @param termWeightingDocFun The term weighting function to apply on
    the binary query
#' @param lsam The LSA matrix decomposition after SVD
#' @return The original string query (q),
#' the binary query document in the latent space (bin_q),
#' the query document after tf-idf weighting (w_q),
#' the query document in the latent space (ls_q = dls = V * Sigma)
    and its normalization (lsn_q),
#' the query document equivalent to V matrix rows (dk = V),
#' the V * Sigma^1/2 element for semantic similarity calculation
    between query and terms (dksrs)
makeQuery <- function(q, tdm, termWeightingDocFun, lsam) {

  # THEORY
  # uk * sigmak * vk_t
  # -> uk (U) = matrix terms x latent variables
  # -> vk (V) = matrix documents x latent variables

  # LSA PACKAGE
  # lsam$tk * lsam$sk * lsam$dk
  # -> lsam$tk = uk (U)
  # -> lsam$dk = vk (V)
  # -> dls = lsam$dk %*% diag(lsam$sk) = vk * sigmak = V * Sigma

  # Create the query vector (binary vector)
  # Transform the query in a vector representing the presence/absence
  # of each term of the bag of words representation
  bin_q <- query(q, rownames(tdm))

  # The query is like a new document to add to the latent space
  # So it applies all the transformations made to those already inside
  # (term weighting, normalization) in order to fold it into the LSA
     space
  # The query vector is now equivalent to a column of tdm matrix
  w_q <- termWeightingDocFun(bin_q, tdm)
  # Calculate the position of the query in the latent space (V *
      Sigma)
  ls_q <- t(w_q) %*% lsam$tk

  # Calculate the normalized query vector in latent space for
      visualization purposes
  lsn_q <- normRows(ls_q)

  # Transform the query vector in a new document (row of V matrix)
  # q_k = q^T * U_k * Sigma_k^-1
  dk <- ls_q %*% diag(lsam$sk ^ -1)

  # The similarity between a query and some terms is calculated as a
      cosine similarity
  # considering the V representation for the query and the terms
      vectors multiplicated
  # by Sigma^1/2.
  # cosine(V * Sigma^1/2, U * Sigma^1/2) = cosine(dk %*%
      diag(sqrt(lsam$sk), lsam$tk %*% diag(sqrt(lsam$sk))
  # dksrs is so one of the two necessary elements for similarity
      calculation
  dksrs <- dk %*% diag(sqrt(lsam$sk))

  # Return a named list with the results
  list(q = q, bin_q = bin_q, w_q = w_q, ls_q = ls_q, lsn_q = lsn_q,
      dk = dk, dksrs = dksrs)

}
```

Code 4.3: Function makeQuery implementation.

4.4 Store triples

Once triples are generated, the graph needs to be updated in the database. Stardog has published a node package[5] which ease the query creation and execution. Before running the query, it's important to recognize if triples contain well known terms and use their definition instead of creating new ones. Therefore for each term, a query is run on the graph and in case a match is found its prefix is used instead. Given a triple, like cat $\rightarrow$ associated_with $\rightarrow$ animal, the procedure is:

[5]https://www.npmjs.com/package/stardog

1. Check for entities existence (cat, associated_with, animal)
    - Create a new entity in case it doesn't exist with the used prefixes
    - Use the existing entity in case it's already present
2. Create a SPARQL query
3. Run the query

The triples are now stored in the database and can be therefore queried, with or without enabling the reasoning.

The class which implements this logic is the `StardogGraphPersister`. It establishes the connection to the database when it's initialized, builds the query to insert the new triples and runs it.

```
const databaseName = config.env.databaseName;

export default class StardogGraphPersister implements
    GraphPersister {
  private readonly connection: Connection;
  private static recognizedTerms: Map<string,string>;

  constructor(
    username: string = 'admin',
    password: string = 'admin',
    endpoint: string = 'http://localhost:5820',
  ) {
    this.connection = new Connection({ username, password,
        endpoint });
    StardogGraphPersister.checkConnection(this.connection);
  }

  public persistTriples(triples: Array<GraphTriple>) {
    StardogGraphPersister.checkConnection(this.connection);
    query
      .execute(
        this.connection,
        databaseName,
        StardogGraphPersister.createInsertTriplesQuery(triples),
        'application/sparql-results+json',
        {
          limit: 10,
          offset: 0,
        },
      )
      .then(({ body }: any) => {
        logger.info(JSON.stringify(body));
      })
      .then(() =>
        logger.info(
          `Persisted ${triples.length} triples into stardog
              database.`,
        ),
      )
      .catch((reason) => logger.info(`ERROR ${reason}`));
  }

  private static tripleToQueryFormat(triple: GraphTriple) {
    return `${triple.getFirst()} ${triple.getSecond()}
        ${triple.getThird()} .`;
  }

  private static createInsertTriplesQuery(triples:
      Array<GraphTriple>) {
    const ret = `INSERT DATA { ${triples
      .map(StardogGraphPersister.useRecognizedTerms)
      .map(StardogGraphPersister.tripleToQueryFormat)
      .join(' \n')} }`;
    logger.info(ret);
    return ret;
  }

  private static useRecognizedTerms(triple: GraphTriple):
      GraphTriple {
    return new GraphTriple(
        recognizedTerms[triple.getFirst()] || triple.getFirst(),
        recognizedTerms[triple.getSecond()] || triple.getSecond(),
        recognizedTerms[triple.getThird()] || triple.getThird(),
    );
  }
}
```

Visually, the insertion of a triple corresponds to the creation of newly specified nodes and of the edge specified. An example has been created in order to illustrate how the graph would change subsequently to the insertion of a triple. We're now pretending that our application is connected to a text source where information about artists and music band are shared in real time.

The starting situation of the database, focused on a single music band, could be queried in stardog using SPARQL syntax like in Code 4.4, and the result would look like Figure 4.2.

In case new text like “Have you seen what’s in the new? The Red Hot have a new member, his name is Samuele Ceroni!” is retrieved by the text provider, it would be processed using the POIROT methodology and a new triple like :Samuele_Ceroni → :memberOf → :Red_Hot_CHili_Peppers might be generated.

The web application would therefore generate the Code 4.5 in order to update the knowledge of the graph.

Finally, running again the query in Code 4.4 would show the up-to-date situation, which is like showed in Figure 4.3.

```
CONSTRUCT {
    ?subject ?predicate ?object
}
WHERE {
    ?subject ?predicate ?object .
    filter (?predicate != rdf:type
         && ?predicate != rdfs:domain
         && ?predicate != rdfs:range
         && ?predicate = :memberOf
         && ?object = :Red_Hot_Chili_Peppers
         )
}
```

Code 4.4: SPARQL query to show a music band in Stardog.

```
INSERT DATA {
    :Samuele_Ceroni :memberOf :Red_Hot_Chili_Peppers
}
```

Code 4.5: SPARQL query to insert a member of a music band in Stardog.

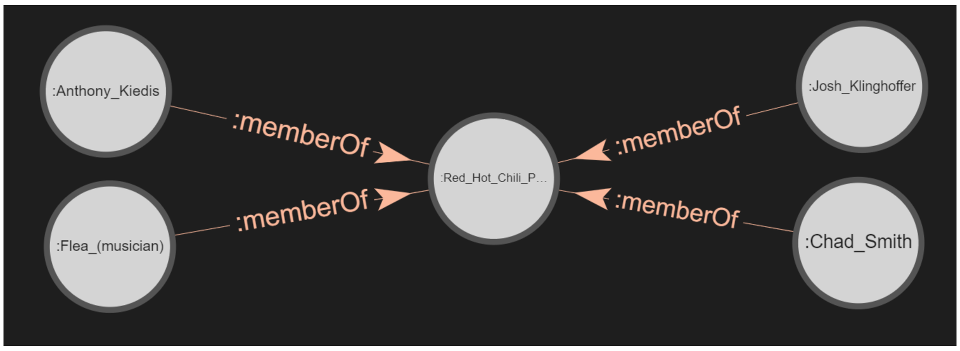

Figure 4.2: Stardog query before triple insertion.

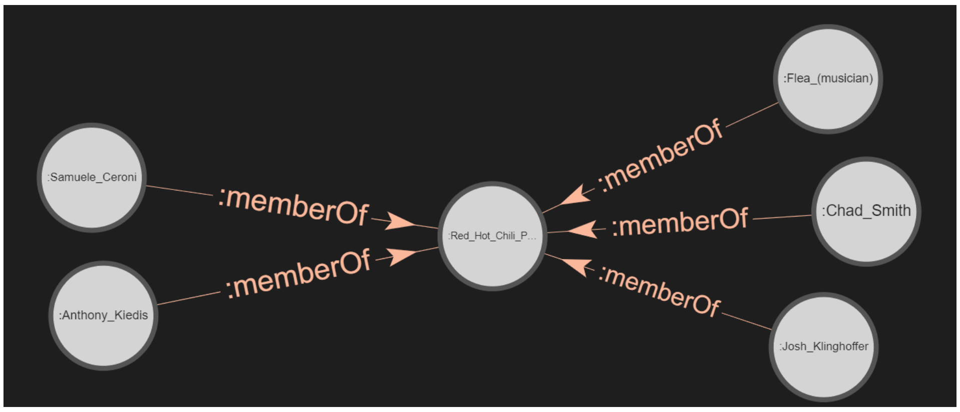

Figure 4.3: Stardog query after triple insertion.

Chapter 5

Conclusions and further developments

This thesis has shown a concrete contribute in one of the most challenging mission the world have these days: convert data into information. It also makes an impact on the case of study: the Esophageal Achalasia, keeping more accurate and always up-to-date information and putting in the first line the voice of patients. Through the usage of one of the most common web server technologies and one of the most used and high-performance database engine, a new service to keep knowledge aligned with new posts on the group has been created.

As future developments, this service could become a real application available on the web to automatically create a knowledge graph from multiple text sources and keep it up-to-date. First of all, instead of an architecture polling-based which checks for updates on text providers every once in while, a webhook-based application would be extremely more efficient, especially if the application needs to scale. At the moment the domain is limited to a rare disease, but the same application could ideally be able to work with multiple domains and multiple schema on the database. Furthermore, multiple text sources would expand dramatically the precision of the data, and ideas of implementable interfaces (other than *FacebookTextProvider*) would be RDF Site Summary (RSS) support, *Twitter*, *Instagram* and other social media. Lastly, but probably the most important, find out a way to use information gathered and organized in the knowledge graph. With that purpose, a classifier component would help in modeling and recognizing recurring pattern in habits of patients, reasoning on data. Ultimately, the graph could definitely be used to create a chat bot to respond or to suggest posts to people that are looking for specific information already provided in the past in the same channel.

Bibliography

[1] DB-Engines Ranking - Trend of AllegroGraph vs. Grakn vs. Neo4j vs. Stardog Popularity. `https://db-engines.com/en/ranking_trend/system/AllegroGraph%3BGrakn%3BNeo4j%3BStardog`. [Online; accessed 23-February-2021].

[2] Giacomo Frisoni and Gianluca Moro. Phenomena explanation from text. In *Data Management Technologies and Applications - 9th International Conference, DATA, Revised Selected Papers*, Communications in Computer and Information Science, pages 1–24. Springer, 2021. Accepted as regular paper.

[3] Giacomo Frisoni, Gianluca Moro, and Antonella Carbonaro. Unsupervised descriptive text mining for knowledge graph learning. In Ana L. N. Fred and Joaquim Filipe, editors, *Proceedings of the 12th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2020, Volume 1: KDIR, Budapest, Hungary, November 2-4, 2020*, pages 316–324. SCITEPRESS, 2020.

[4] John F. Sowa. *Knowledge representation: logical, philosophical, and computational foundations*. Brooks/Cole, 2000.

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

[6] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. In Michael Martin, Martí Cuquet, and Erwin Folmer, editors, *Joint Proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems - SEMANTiCS2016 and the 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS'16) co-located with the 12th International Conference on Semantic Systems (SEMANTiCS 2016), Leipzig, Germany, September 12-15, 2016*, volume 1695 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[7] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Trans. Knowl. Data Eng.*, 29(12):2724–2743, 2017.

[8] Dominik Tomaszuk, Renzo Angles, and Harsh Thakkar. PGO: describing property graphs in RDF. *IEEE Access*, 8:118355–118369, 2020.

[9] Dominik Tomaszuk and David Hyland-Wood. RDF 1.1: Knowledge representation and data integration language for the web. *Symmetry*, 12(1):84, 2020.

[10] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. In *International semantic web conference*, pages 30–43. Springer, 2006.

[11] Jans Aasman. Property Graphs are not enough. `https://allegrograph.com/property-graphs-are-not-enough`. [Online; accessed 22-February-2021].

[12] José Emilio Labra Gayo, Eric Prud'hommeaux, Iovka Boneva, and Dimitris Kontokostas. *Validating RDF Data*. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers, 2017.

[13] Florian Bauer and Martin Kaltenböck. Linked open data: The essentials. *Edition mono/monochrom, Vienna*, 710, 2011.

[14] Linked Open Data Cloud. `https://lod-cloud.net/`. [Online; accessed 27-February-2021].

[15] N Senthilselvan, V Subramaniyaswamy, and KR Sekar. Information retrieval for biomedicine applications through linked open data based optimization model. 2017.

[16] Ann-Kristin Kock-Schoppenhauer, Christian Kamann, Hannes Ulrich, Petra Duhm-Harbeck, and Josef Ingenerf. Linked data applications through ontology based data access in clinical research. *Studies in health technology and informatics*, 235:131–135, 2017.

[17] Boris Villazón-Terrazas, Luis M Vilches-Blázquez, Oscar Corcho, and Asunción Gómez-Pérez. Methodological guidelines for publishing government linked data. In *Linking government data*, pages 27–49. Springer, 2011.

[18] Nigel Shadbolt and Kieron O'Hara. Linked data in government. *IEEE Internet Computing*, 17(4):72–77, 2013.

[19] Wang TianXiang. A survey on temporal knowledge graph embedding. 2020.

[20] Orphanet. The portal for rare diseases and orphan drugs. `https://www.orpha.net`. Accessed 9 Mar 2021.

[21] Slade A., Isa F., Kyte D., Pankhurst T., Kerecuk L., Ferguson J., Lipkin G., Calvert M. Patient reported outcome measures in rare diseases: a narrative review. *Orphanet Journal of Rare Diseases*, 13:61, April 2018.

[22] GlobalGenes. Rare diseases: facts and statistics. `https://globalgenes.org/rare-facts/`. Accessed 9 Mar 2021.

[23] EURORDIS. The voice of rare disease patients in Europe. `https://www.eurordis.org/about-rare-diseases`. Accessed 9 Mar 2021.

[24] Kuiper G., Meijer O.L.M., Langereis E.J., Wijburg F.A. Failure to shorten the diagnostic delay in two ultra-orphan diseases (mucopolysaccharidosis types I and III): potential causes and implications. *Orphanet Journal of Rare Diseases*, 13:2, January 2018.

[25] Shire. The Global Challenge of Rare Disease Diagnosis. `https://www.shire.com/-/media/shire/shireglobal/shirecom/pdffiles/patient/shire-diagnosis-initiative-pag-leaflet.pdf`. Released in Feb 2016.

[26] Giacomo Frisoni. A new unsupervised methodology of Descriptive Text Mining for Knowledge Graph Learning. Master thesis, University of Bologna, 2020.

[27] Giacomo Frisoni, Gianluca Moro, and Antonella Carbonaro. Learning interpretable and statistically significant knowledge from unlabeled corpora of social text messages: A novel methodology of descriptive text mining. In Slimane Hammoudi, Christoph Quix, and Jorge Bernardino, editors, *Proceedings of the 9th International Conference on Data Science, Technology and Applications, DATA 2020, Lieusaint, Paris, France, July 7-9, 2020*, pages 121–132. SciTePress, 2020.

[28] Thomas K Landauer and Susan T Dumais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological review*, 104(2):211, 1997.

[29] Giacomo Domeniconi, Gianluca Moro, Andrea Pagliarani, Karin Pasini, and Roberto Pasolini. Job recommendation from semantic similarity of

linkedin users' skills. In *International Conference on Pattern Recognition Applications and Methods*, volume 2, pages 270–277. SCITEPRESS, 2016.

[30] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[31] Thomas Hofmann. Probabilistic latent semantic analysis. *arXiv preprint arXiv:1301.6705*, 2013.

[32] Virginia Klema and Alan Laub. The singular value decomposition: Its computation and some applications. *IEEE Transactions on automatic control*, 25(2):164–176, 1980.

[33] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

[34] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[35] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[36] DB-Engines Comparison - AllegroGraph vs. Grakn vs. Neo4j vs. Stardog. `https://db-engines.com/en/system/AllegroGraph%3BGrakn%3BNeo4j%3BStardog`. [Online; accessed 10-March-2021].