Ministry Of Education Of The Azerbaijan Republic

UFAZ

LITIS

Ali Khudiyev

A thesis presented for the degree of

Master of Science

Knowledge Formalization With Domain Experts In The Loop

Developing OTTR Templates

Speciality: Data Science and Artificial Intelligence

Supervisors: Prof. Cecilia Zanni-Merk, Ph.D. Matthias Sesboüé

Applied Computer Science

French-Azerbaijani University

Azerbaijan

21 February 2022 - 27 June 2022

Abstract

The use of ontologies has already been shown to be helpful and beneficial in several different domains such as search engines, recommender systems, adaptive controlling, and so on. Knowledge formalization is one of the main processes used for converting tacit knowledge to explicit knowledge for further use by the machines with the primary goal of yielding autonomy. Several aspects of formalizing human knowledge are discussed in this paper as well as a brief introduction to O/ontology, its applications in Computer Science, and the potential benefit of formalizing knowledge in the domain of search engines. The main goal of this project is to automate the knowledge acquisition process by creating an ontology and building a knowledge graph (KG) in the domain of a search engine. I also describe an already-existing database structure and propose methodologies to ultimately improve search engine efficiency and/or quality through knowledge acquisition. Building a KG is an essential first part of making the data more semantically accessible and searchable by the search engines. Ontological concepts and relations as well as software architecture and different programming paradigms are discussed thoroughly in the paper by sharing important implementation details. Multiple UML diagrams and demonstrative examples are also given for better readability with the least amount of ambiguity. Final (experimental) results as well as discussions about the robustness and efficiency of the architecture and implemented algorithms are shared. Finally, I share my conclusions on the use of ontologies or knowledge graphs and the use of OTTR language to build them.

Referat

Ontologiyaların istifadəsinin axtarış motorları, tövsiyə sistemləri, adaptiv nəzarət və s. kimi bir neçə fərqli domendə faydalı olduğu artıq sübut edilmişdir. Biliyin rəsmiləşdirilməsi, əsas məqsədi muxtariyyət əldə etmək məqsədi daşıyan maşınların sonrakı istifadəsi üçün gizli biliyin açıq biliyə çevrilməsi üçün istifadə olunan əsas proseslərdən biridir. Bu yazıda insan biliyinin rəsmiləşdirilməsinin bir neçə aspekti, həmçinin O/ontologiyaya qısa giriş, onun Kompüter Elmində tətbiqləri və axtarış motorları sahəsində biliklərin rəsmiləşdirilməsinin potensial faydası müzakirə olunur. Bu layihənin əsas məqsədi ontologiya yaratmaq və axtarış sisteminin domenində bilik qrafikini (KG) qurmaqla biliklərin əldə edilməsi prosesini avtomatlaşdırmaqdır. Mən həmçinin artıq mövcud olan verilənlər bazası strukturunu təsvir edirəm və bilik əldə etməklə axtarış sisteminin səmərəliliyini və/və ya keyfiyyətini son nəticədə yaxşılaşdırmaq üçün metodologiyalar təklif edirəm. KG-nin yaradılması məlumatların semantik cəhətdən daha əlçatan və axtarış motorları tərəfindən axtarıla bilən olmasının vacib ilk hissəsidir. Ontoloji anlayışlar və əlaqələr, eləcə də proqram arxitekturası və müxtəlif proqramlaşdırma paradiqmaları məqalədə mühüm icra təfərrüatları paylaşılmaqla hərtərəfli müzakirə olunur. Ən az qeyri-müəyyənliklə daha yaxşı oxunaqlı olmaq üçün çoxsaylı UML diaqramları və nümayiş etdirici nümunələr də verilmişdir. Yekun (eksperimental) nəticələr, eləcə də arxitekturanın möhkəmliyi və səmərəliliyi və həyata keçirilən alqoritmlər haqqında müzakirələr paylaşılır. Nəhayət, mən ontologiyaların və ya bilik qrafiklərinin istifadəsi və onları qurmaq üçün OTTR dilinin istifadəsi ilə bağlı nəticələrimi bölüşürəm.

Аннотация

Использование онтологий уже показало свою полезность и выгоду в нескольких различных областях, таких как поисковые системы, рекомендательные системы, адаптивное управление и т.д. Формализация знаний является одним из основных процессов, используемых для преобразования неявных знаний в явные знания для дальнейшего использования машинами с основной целью обеспечения автономии. В этой статье обсуждаются несколько аспектов формализации человеческих знаний, а также краткое введение в Онтологию, ее приложения в компьютерных науках и потенциальные преимущества формализации знаний в области поисковых систем. Основной целью данного проекта является автоматизация процесса получения знаний путем создания онтологии и построения графа знаний (ГЗ) в предметной области поисковой системы. Я также описываю уже существующую структуру базы данных и предлагаю методологии для окончательного повышения эффективности и/или качества поисковых систем за счет приобретения знаний. Создание KG является важной первой частью повышения семантической доступности данных и поиска их поисковыми системами. В документе подробно обсуждаются онтологические концепции и отношения, а также архитектура программного обеспечения и различные парадигмы программирования с указанием важных деталей реализации. Несколько диаграмм UML и демонстрационные примеры также приведены для лучшей удобочитаемости с наименьшим количеством двусмысленности. Публикуются окончательные (экспериментальные) результаты, а также обсуждения надежности и эффективности архитектуры и реализованных алгоритмов. Наконец, я делюсь своими выводами об использовании онтологий или графов знаний и использовании языка OTTR для их построения.

Acknowledgements

I would like to thank my supervisors Prof. Cecilia Zanni-Merk, Ph.D. Matthias Sesboüé for letting me use my own creativity and propose solutions to the encountered problems during the internship. I would like to also thank Postdoc Mathieu Bourgais, Ph.D. Matthieu Bellucci for interesting discussions about AI, explainability, knowledge graphs, etc.

I would like to thank Prof. Benoit Gauzere for sharing his insights and intuitions based on his previous experiences during the interview process and others with whom I shared internship and life experiences.

Besides the LITIS laboratory I would like to thank some professors and assistant professors at UFAZ with whom I had very intriguing and insightful lectures and/or sessions.

List of Figures

List of Tables

Contents

1 Introduction

Search engine is a software program used to fetch information from a database by providing textual description. Nowadays, it is common to see and use different search engines on the web and receive the desired information as Search Engine Results Pages (SERPs) [1]. SERPs can be based on two types of searches: *organic* and *sponsored*. Organic search is not affected by any external resources such as advertisements displayed on the web pages whereas sponsored search ranks documents based on such resources. There are three main processes run by any search engine: *web crawling* [2], *indexing* [3] and *searching* [4].

Web crawling is a technique that requires an internet bot or spiderbot to go from site to site and exploit them individually by the guidance of the "*robots.txt*" file provided within the website. This file usually contains information about the inner structure of the site by providing a list of searchable and non-searchable directories.

Indexing is used for associating a set of tokens from the web pages to their domain names. This process makes the associations available to the public for them to be able to find the most suitable web resources (i.e., web pages) for their needs. Thanks to the indexing process, users can use one or more words to describe the desired resource.

Finally, the searching process takes care of the rest; the SERPs are usually ranked according to their indices, and relevance before further lookups, reconstruction and markup of the snippets to display the matched tokens. However, search engines are not only limited by the processes mentioned above. They can provide advanced techniques for the users such as *proximity search* [5], *concept-based search* [6], and so on. Proximity search is a feature that allows the user to specify the distance between

the provided tokens which can be measured by intermediate tokens appearing on the web pages. On the other hand, concept-based search involves statistical analysis of the tokens and/or phrases provided in the user queries on the web pages.

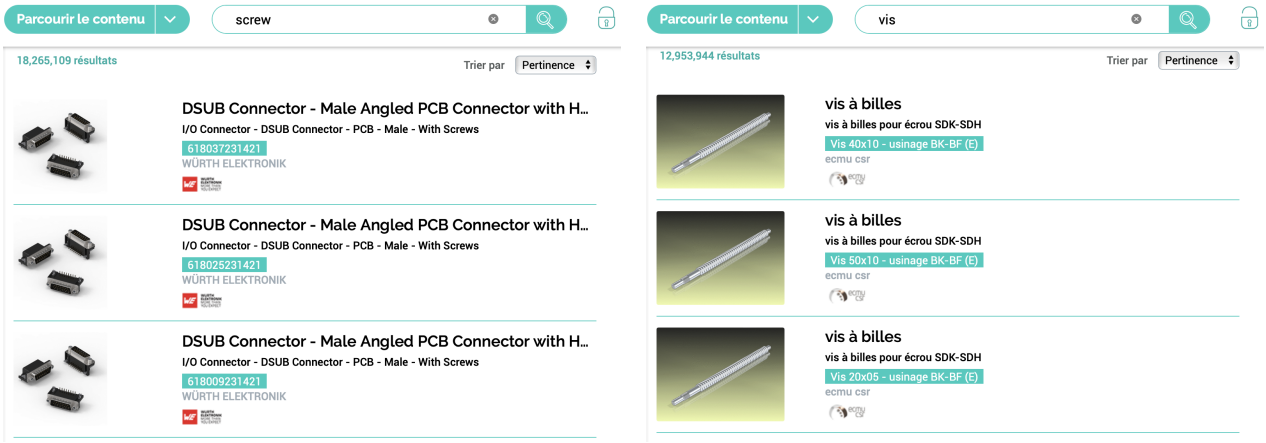
Ontology is a branch of philosophy whose main goal is to define a *being* or *thing*. Although philosophers usually use different methodologies to come up with the answers, Ontology can be used with several modifications in Computer Science. A similar field in CS is usually written as *ontology* with the first letter being small. There are three types of ontologies known as *domain ontology*, *upper ontology* and *hybrid ontology*. Domain ontologies focus on the concepts and relations within some domain while upper ontologies try to be common shared vocabulary across a wide range of domain ontologies. On the other hand, hybrid ontologies are a combination of both upper and domain ontologies. Ontology engineering is another field that focuses on different strategies and paradigms for building ontologies. Some of the main concepts that the field describes include different levels of abstractions, general approaches to vocabulary development, evaluating ontologies, ontology design patterns, term excerption and development, terminology analysis and curation, conceptual modeling [7, 8].

Nowadays, it is not rare to see AI methodologies and ontologies working together. The main benefit of combining these two fields is due to the fact that we can create explainable systems. Most classical AI systems are considered black boxes which are hard to debug and understand the reason(s) behind their high-level decisions in a *reasonable time*. Explainable Artificial Intelligence (XAI) [7] is helpful in such situations. Another benefit of using ontologies with black-box models is that the developed methods can be guiding furthermore to explore the search space more efficiently. For example, if we just tried to brute-force win the chess game played against a human opponent, we would run out of time limit because the number of possibilities for exploration is very vast. Instead of blindfolded brute-force, we could instead try to eliminate the least promising cases just like the human opponent would typically do. This is essentially known as heuristic guided search and in the case of

chess, one type of such algorithms is obtained by applying alpha-beta pruning to the minimax algorithm. Returning back to the ontologies, they can also be used to guide the searching process to increase performance. Although using heuristics and ontologies may seem similar, the main distinction between them is that heuristics need not be closely tied to the concepts and relations as ontologies do and they are developed in the context of some objective function, and therefore, can be changed completely or modified partially depending on it while ontologies are there to solely represent concepts and relations.

The main goal of the project, on whose part I have been working, is to improve the search engine of a company called **Traceparts**[1], and to achieve this goal, the approach taken is by providing semantic search capabilities to the already existing search engine of the company. To be able to do that, we introduce semantic layers on top of which a unified knowledge graph is constructed by using the relational database of the company. Although the database contains a vast amount of entities, none of them is linked to each other in a sensible way and therefore, improvements on the carried search operations by the current search engine of the company are limited. By linking the entities and maintaining a common ground for their types and relationships with each other, the search process can be optimized to be more efficient and satisfying for the end-users. Currently, TraceParts' search engine only implements text-based lookups on the ElasticSearch database by matching the words or tokens provided in the user query. Figure 1.1 visually demonstrates this.

[1] `https://www.traceparts.com/en`



(a) SE queried with "screw" (b) SE queried with "vis"

Figure 1.1: Example queries on TraceParts SE

Although ElasticSearch is well-known for its capabilities for fast textual searchess across the database entities, we believe that it is not an efficient use of the database to throw a bunch of words at it, which can be typed in different languages or formats according to different standards, and expect it to fetch satisfactory results. Such a search engine would fail to provide useful and proper web resources in so many easily imaginable cases shown below:

1. What if the database contained the word "screw" but not "vis"[2]?

2. What if the database contained screws but with no proper understanding of the dimensionality of screws?

3. What if the user query was like "screwdriver used for small size screws" and the database had no understanding of whether the user was looking for wrenches, screws, or everything which can be described as small-sized?

The issues mentioned above are the by-product of performing only raw text-based searches across the database entities. It is not hard to imagine that some of the entities may contain the same words such as "small", "medium", "large", "dimension", "standard", "colored" and so on.

[2] "La vis" means a screw in French.

Currently, the search process is blindfolded, meaning that it has no idea what to look for in terms of concepts that matter to the user. To avoid such inefficiencies in the run-time performance and ranking of the search engine result pages, we can guide the search process by using ontologies and/or knowledge graphs. Knowledge acquisition plays a crucial role in the process of developing ontologies and/or knowledge graphs. Several approaches to address some of the issues related to knowledge acquisition bottlenecks as well as information retrieval systems and search engine optimization strategies are demonstrated in chapter 3. In this thesis work, several knowledge acquisition strategies are described for introductory purposes for the following sections. My main contributions in this project include the development of a software architecture, implementation of different processing paradigms, and building a knowledge acquisition pipeline all of which are introduced and discussed thoroughly in chapter 4. To acquire knowledge from the relational database of TraceParts, a relatively simple ontology has been developed and used to build the ultimate knowledge graph. The ontology and knowledge graph building process have been described in a more detailed fashion later on in the same chapter. In chapter 5, implemented library and the final software as well as several tests are discussed. Finally, I share my conclusions on this project and the internship experiences in chapter 6.

The goal of this project work has been dedicated to develop a PoC system with which we could test different hyphothesis and compare benchmark results relatively quickly and easily. Such a system would play the foundational role in the ultimate search engine version and be sufficient for further improvements step-by-step, iteratively.

2 About LITIS Laboratory

LITIS is a laboratory of the University of Havre Normandy, INSA Rouen Normandy and University of Rouen Normandy. The laboratory is a member of "Digital Normandy" which is the Norman network and the doctoral school of MIIS. LITIS is also a partner of the Normastic CNRS Research Federation. There are several research fields in LITIS which are shown below:

- Information access
- Ambient intelligence
- Biomedical information processing

Beside the above shown fields of research, there are also several applications that are being addressed to by the work of laboratory workers:

- Automotive and smart territories
- Information acces in all sectors
- Health

There are several departments in the laboratory each of which has expertise and works on different fields and aims at professional quality results. These fields are as shown below:

- **Machine Learning**

  Different techniques and/or models such as Markovian models, graph-based

classification, kernel machines, etc. are used in ML applications. Understanding and being able to use relevant strategies is not a trivial job and the laboratory has theoretical and algorithmic expertise on these domains.

- **Intelligent vehicle**

  Transportation can become even efficient with the help of intelligent vehicles and other tools used in everyday transit. Big data management for such intelligent transportation systems plays an important role for this field of research.

- **Multi-agent systems**

  Use of AI and Semantic Web technologies in order to automate decision-making processes through reasoning and explainability is the main goal. This research field also focuses on the human-machine interfaces for the development of socio-technical systems.

- **Health and Information Technology**

  Analyzing biologcial data to gather relevant information in the field of bioinformatics. Therapeutic monitoring and prediction in the domain of medical imaging is also a part of this research field.

- **Combinatorics and Algorithms**

  Having many applications in information processing such as words, automatons, free monoids, etc., the members of this field focus on the study of models of algebraic nature to analyze combinatorial and algorithmic aspects of them.

Besides scientific aspects and achievements of the LITIS laboratory, there are safety guidelines for emergency and those guidelines are practiced by the workers from time to time. In LITIS, researchers are very welcoming and they usually try to help each other on local social media channels when there is some technical problems to be solved. In my opinion, such attitude is one of the biggest achievements of the laboratory or even any company.

3 Literature Review

Information Retrieval (IR) is a process of obtaining relevant information from a database according to the given query. User queries are usually provided in a text format, however, there is no theoretical limit or restriction for other formats. IR systems should not be confused with database systems; although they perform searching processes on entities(i.e., documents), the key difference remains in the ranking process. Ranking documents to compute their relevancy is required by the IR system and therefore, it may be the case that the system will output results that do not match the query. In contrast, classical databases search for and output documents that always match the given query.

Different models can be used by IR systems. Although the strategies differ, the main goal is always to find the most relevant information. To achieve this goal, the models transform documents into an appropriate representation and this is where different models can utilize different representations. There are three types of models which are shown below:

1. *Set-theoretic*. Models of this type use the set representation to describe documents; a document can be viewed as a set of phrases, words or even letters. Operations used to find similarities are set operations.

2. *Algebraic*. Such models usually use vector or matrix representation for documents and the similarity between the given query and a document is a scalar value yielded by some relevant algebraic operation(i.e., dot product for vectors).

3. *Probabilistic*. The core idea behind these models is the probabilistic inference

used for document retrieval. Baye's theorem [9] plays a very important role for such models. Similarities are computed as a probability for the relevancy of a document if given a query.

Some of the most popular search engines that have been able to utilize IR systems more than others are *Google*, *YouTube*, *Amazon*, *Microsoft Bing*, and so on. However, such big companies usually develop relatively very complex systems that are the best fit for very large-scale problems. Although most of such systems are commercially available, it may be too much of an overhead to try to adapt and use them in small or medium-scale projects. Instead, one should try to observe the key concepts and come up with specialized methodologies or strategies which will maximize the expected outcome/value of the project.

Search Engine Optimization (SEO) is a process of efficiency of a search engine (SE) or website traffic by applying a set of methods such as indexing, crawling, cross linking, etc [10]. There are different SEO methods used to attract more and more users to websites by providing more appropriately ranked content.

There are many SEs that consider little or no semantics to find relevant results for a given user query; most of these systems perform text-based searches across the database. Adding semantics in order to guide the search algorithms is not a new idea and has been implemented by many researchers. Lai et al. have developed a fuzzy ontology for their fuzzy search engine Fuzzy-Go [11]. Their ontology uses fuzzy logic to represent semantic distances between tokens(i.e., keywords, words) which enables Fuzzy-Go to go beyond text-only search by also being able to consider the synonyms of terms.

Modern SEs index contents of web pages by searching and matching the resources with the user query which is usually enriched by one or more Information Retrieval (IR) techniques. Some of the well-known IR techniques are the TF-IDF ranking model [12] and PageRank [13]. Bonino et al. have proposed an automatic search refinement mechanism based on ontology navigation for which computing

semantic query refinement requires focalization and generalization [14].

Knowledge acquisition has always been a bottleneck for large-scale systems. Since such systems require a significant amount of human expert knowledge to operate, scalability becomes a problem. Tudorache et al. have developed a lightweight web-based knowledge acquisition tool and ontology editor called WebProtégé [15]. They have adopted the infrastructure of Protégé which enables collaboration on the back-end and using Google Web Toolkit on the front-end. Although it is useful to have such knowledge acquisition tools, one downside is usually an intermediate interface through which human experts convey their knowledge. Currently, experts are required to have some degree of understanding in FoL[1] [16] to be able to express their knowledge of some domain. A subset of FoL is used by ontologies due to their reasoning capabilities [17].

[1] First-order logic

4 Data and Methods

4.1 Data

Only a limited amount of data has been provided by the company *TraceParts* due to confidentiality. However, provided data has been enough to develop a methodology for, at least, a proof of concept. To be more specific about the provided dataset, 100 database instances have been accessed and loaded into the local machine environment. For this purpose, docker container(s) has been used to abstract the database service(s).

ElasticSearch (ES) is an open-source RESTful search and analytics engine which works in a distributed manner under the hood. It has been developed with Java and is based on Apache Lucene. In the ES database, data is stored with corresponding indices, and to retrieve it, we can send queries to its "_search" endpoint.

4.1.1 Database Schema

The database has a somewhat complicated schema, however, there are several fields of interest for the matter of this internship. These fields include *Part Family*, *Part Number*, *Part Family Name*, and *Part Number Name*.

In the current version of the database used by the company, there are fields with the same values that are repeated in several places in the database memory. Such use of the service is probably inefficient in terms of memory complexity and maybe processing as well. In either case, there is inefficiency in the schema design.

PartFamily	PartNumber	PartFamilyName	PartNumberName
10-05112010-061898	NSYMR34	Telequick Mounting plate	Telequick Mounting plate 300X400
10-07032014-070631	KTC2500ET32B	KT 3X2500CO FEEDER	KT 3X2500CO FEEDER LENGTH
10-07032014-069644	KTC1000lP7C1	KT 5X1000CO FLAT ELBOW N1	KT 5X1000CO FLAT ELBOW N1
10-19032018-083391	LC1D12W7	Contactor	Contactor - LC1D12W7

Table 4.1: Example database instances

4.2 Methods

To make the TraceParts' search engine more semantic, the proposed solutions assume the existence of knowledge graph of database entities where there are relevant and useful semantic linkages between them. Since there is no such KG, I have proposed and implemented several solutions to fulfill this requirement. More specifically, I have worked on the Knowledge Acquisition part. The figure below illustrates the already-existing architecture of Knowledge Acquisition.

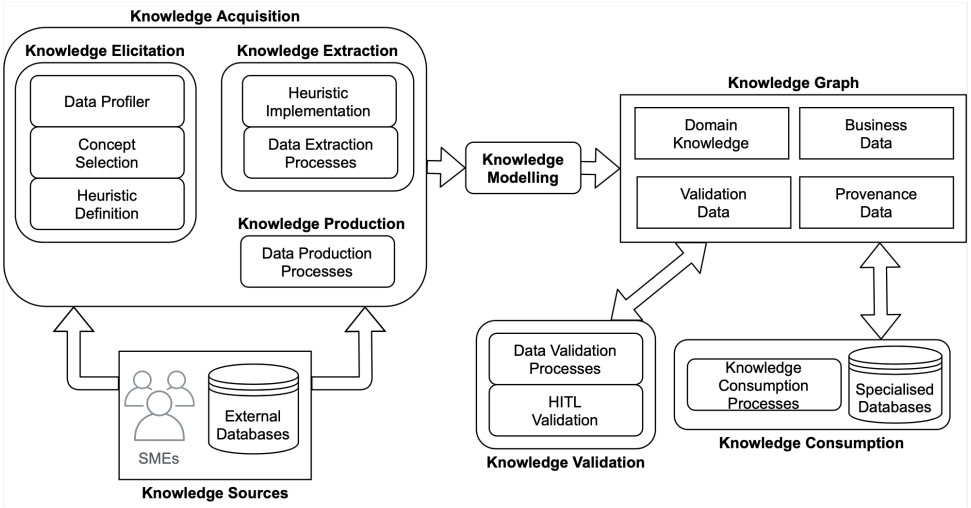


Figure 4.1: Knowledge Acquisition pipeline

In this figure, round-shaped rectangular boxes indicate the processes and sharp-shaped boxes indicate the storage. *Knowledge elicitation*, *knowledge extraction*, and *knowledge consumption* are the three parts of the knowledge acquisition process.

Knowledge elicitation is the process of acquiring tacit knowledge from domain experts in the formal form. Conversion of tacit knowledge or insights into tangible and structured data is not trivial work and needs to be dealt with at the very first step of the project. Knowledge extraction is the process of extracting knowledge from (un)structured data containers. As an example of such a process, *Extract-Transform-Load* (*ETL*) can be given where data is extracted from one or more databases, then transformed in some specific way to be suitable for further use, and finally, loaded into one or more databases. Knowledge production and consumption processes define how the linked data points are produced and requested by the end-users respectively.

Knowledge validation is the process of KB[1] verification and evaluation [18, 19]. Knowledge verification focuses on the structural correctness of knowledge bases whereas evaluation takes care of the inference part so that only correct conclusions can be made within the knowledge base. There are several approaches for KB verification that are based on dependency tables, decision trees, machine learning, graphs, etc., and another set of approaches for evaluation is aimed at refinement, testing, generating, and so on.

Knowledge sources are where we acquire knowledge. These sources can either be subject-matter experts (SMEs) or some external databases. In the case of this project, we have used TraceParts' ElasticSearch database to extract knowledge, transform it to make it comply with the developed ontology, and load it into the Jena graph database. Throughout the rest of the processes, the only knowledge source has been the KG that we have acquired. This also helps us to simplify the knowledge validation steps since we only consider our KG as a unified source of truth and every other process, if needed, is built on top of this single knowledge source.

[1] Knowledge Base

A knowledge graph is a data structure that represents some specific piece of knowledge by using graphs. In the context of KGs, nodes or vertices of the graph represent the concepts while edges or arcs represent the relationships between them. The distinction between ontologies and knowledge graphs is not still clear among many people. According to some sources, ontologies are only to represent concepts and relationships between those concepts whereas knowledge graphs also incorporate individuals or instances. However, this definition is also controversial among some other sources; for example, an ontology can also incorporate individuals or objects according to [20].

Although there are many debates on definitions of ontologies and knowledge graphs, I think that it may be a bit deviating route to take to define something whose whole purpose, ironically, is to define things. The reason behind this thought is that I do not think that every "thing" is clearly and rigorously definable. For example, even the concept of a "straight line" is not rigorously definable in the sense that its definition would not contain the concept of a straight line, or in other words, if circular definitions were not allowed then defining a "straight line" would not be possible at all and that is the reason it is accepted as a *primitive* which is accepted to have no definition [21]. The deeper reason for such things to happen, I think, is due to the nature of the language itself. We use language to describe or communicate our thoughts, feelings, emotions or experiences and these experiences are not inherently captured by the language itself. In fact, language is invented and learned by humans through an iterative process.

4.2.1 Ontology Concerning The Search Engine

The first important part of knowledge acquisition is to have an ontology describing the concepts and relationships between them in a formal way so that there is an agreed-upon structure of data. Developing an ontology is not an as easy job as it may sound. While designing an ontology, the designer should be aware of the implications of his/her design choices. Moreover, it is not a trivial task to be able to identify

necessary and sufficient ontological objects (i.e, concepts and relationships). An ontology is considered well-designed if it is able to say what it is supposed to say and not say what it is not supposed to say. There is a balance between being less expressive to the degree that one cannot represent what is essential and more expressive to the degree that one can express a lot of things that are not meant to be essential. Either case is not considered good practice, in fact, it is the thin line that one should try to establish for the developed ontology [22].

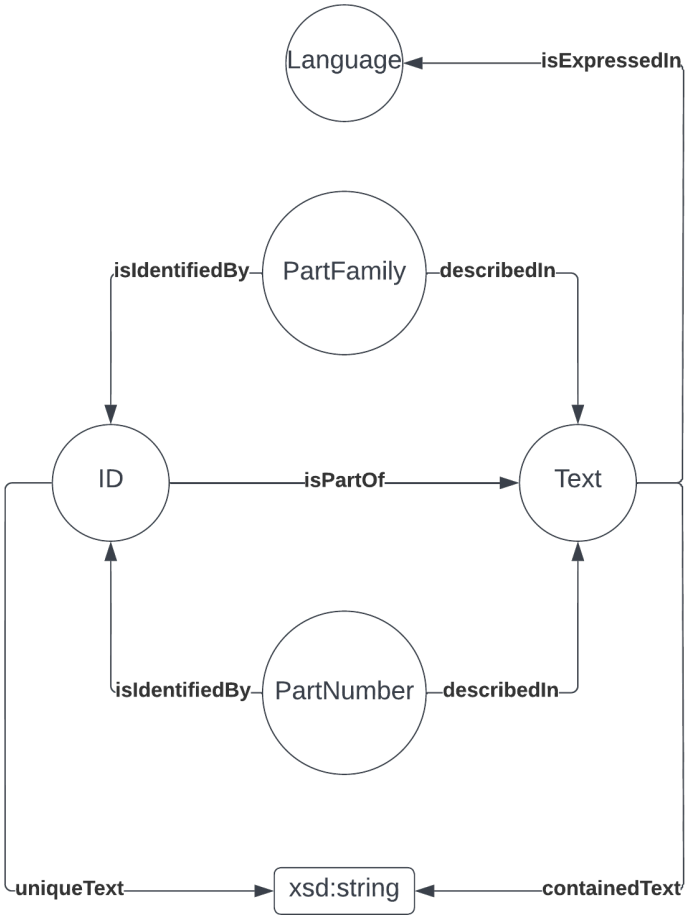


Figure 4.2: Ontology for knowledge acquisition relevant to the TraceParts' SE

In order to build a knowledge graph, one has to have *T-box* and *A-box*. These boxes are the essential parts of any knowledge graph. In the following sections, more specific details about these boxes and what T-box and A-box elements have been defined in this project are described thoroughly (see figure 4.2). Since the reason for building ontologies and/or knowledge graphs is to have a unified vocabulary

throughout the applications, we have also used another ontology called *gist* [23] to integrate some of the concepts defined there that are useful for our purpose. Gist is an upper ontology that consists of business-related concepts with the least amount of ambiguity.

Concepts

Concepts are considered as *terminology box* or *T-box* elements and without them there is nothing to talk about since they define the concepts that can be used in the natural language. One can also think of T-box elements as classes defined in object-oriented programming. Table 4.2 demonstrates the ontological concepts used in the project with their descriptions and source ontology, if any, they have been taken from.[2]

Concept name	**Description**	**Integrated ontology**
ID	Used for unique identification (of Part Family/Number)	gist
Text	Used for textual descriptions (of Part Family/Number)	gist
PartNumber	Indicates a specific CAD model	-(tp)
PartFamily	Indicates a family/set of different CAD models	-(tp)
Language	Indicates the language of a given Text	gist

Table 4.2: Concepts

Relations

Relations and individuals are the elements of *assertion box* or *A-box* and they play an essential role in linking the concepts with each other to get the complete

[2] **tp** refers to the **T**race**P**arts ontology.

knowledge. One can also think of A-box elements as objects declared in object-oriented programming. In table 4.3, relations(i.e., object and data properties), their domains/ranges, and the source ontology they have been taken from are shown.

Relation name	**Domain → Range**	**Integrated ontology**
uniqueText	ID → xsd:string	gist
containedText	Text → xsd:string	gist
isDescribedIn	PartNumber → Text	gist
isIdentifiedBy	PartFamily → ID	gist
isEpxressedIn	Text → Language	gist
isPartOf	ID → Text	gist
hasPartFamily	PartNumber → PartFamily	-(tp)

Table 4.3: Relations

4.2.2 Developing OTTR Template Library

Reasonable Ontology Templates (OTTR) is a specification on which arbitrary ontologies can be created. OTTR is useful due to its capabilities to handle templates.

Templates can be of types either base or non-base. *Base templates* are the ones whose body does not contain any *non-base template* while the templates that are not of the base type can freely contain other non-base and base templates. The difference between the 2 types of templates is that a base template is replaced by its body after the compilation process while a non-base template is continuously replaced by its body until reaching the base template. According to the specification, a template consists of several parts and can be described as below:

```
signature [ parameters ] :: {
        body
}.
```

The template body can contain another template signature with the relevant parameter list and during the compilation process, all the nested templates are exposed

until reaching the base template. Below is shown a part of the developed OTTR library during the internship.

```
tp:createIdInstance [ gist:ID? id,
                      xsd:str ?id_str ] :: {
        ottr:Triple(?id, rdf:type, tp:ID),
        ottr:Triple(?id, gist:uniqueText, ?id_str)
}.

tp:isPartOf [ gist:ID ?id,
              gist:Text ?text ] :: {
        ottr:Triple(?id, gist:isPartOf, ?text)
}.

tp:createTextInstance [ gist:Text ?text,
                        xsd:string ?text_str,
                        gist:Langauge ?language,
                        List<gist:ID> ?ids ] :: {
        ottr:Triple(?text, rdf:type, gist:Text),
        ottr:Triple(?text, gist:isExpressedIn, ?language),
        ottr:Triple(?text, gist:containedText, ?text_str),
        cross | tp:isPartOf(++?ids, ?text)
}.
```

Once we have an OTTR library, we can use it to declare and use instances of our knowledge graph easily without even knowing about the underlying OTTR triples. This also allows us to build knowledge graphs in a more flexible way such that if we needed to change the underlying ontology, we would not have to go through each instance one by one in order to apply this modification throughout the whole graph. Instead, everything would be recompiled with the modified library and that is

it. Even when writing a template library, we tend to build templates as modular as possible so that the code is easily maintainable. This is, in fact, due to the nature of templates and their compilation process. Since each non-base template gets replaced by its body, this process allows us to recursively use a hierarchy of templates which eases the developers' work very significantly.

4.2.3 Software Architecture

There are different paradigms that are used to develop robust software with the help of clear, simple, and intuitive architectures. Such software architectures usually tend to be modular by encapsulating low-level working principles of the software pieces. *Clean architecture* is one of many software architecture design patterns that combine many useful aspects of different paradigms.

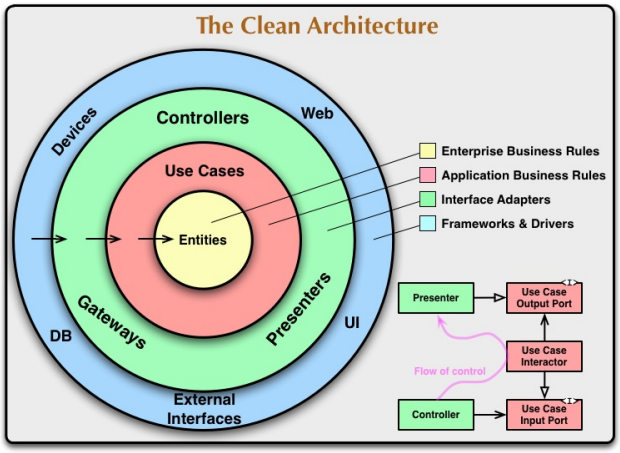


Figure 4.3: The Clean Architecture [24]

Figure 4.3 illustrates the clean architecture visually. Developing clean software architecture requires to comply with one important rule which is called the *dependency rule*. The dependency rule states that any inner layer in the architecture may not depend on any outer layer while the opposite is possible and desired. To fulfill this rule, the most inner layer is called *entities* and is to describe the most general or high-level business rules. Such rules are only business-dependent and have nothing to do with the rest of the developed software. Following this layer, comes the *use cases*. Use cases define the same business rules on the application level which is

to say, such implementation depends on the enterprise-wide business rules. *Interface adapters* shown in the green circle are to serve as some sort of communication medium between the internal and external resources describing data format conversions. Finally, the most outer layer in the architecture is *frameworks and adapters* which include third-party services such as databases, web frameworks, and so on. This layer contains the least amount of code development as well as a potential harm.

Diagrams

Figure 4.4 illustrates the UML class diagram with classes and relations between them. To get the top-down view, the most abstract or high-level classes are **ETL**, **ETLOperation** and **Data**. ETL may use zero or more ETLOperation and each such operation both consumes and produces OperationalData which is a subclass of the Data class. Each ETLOperation object is stored in the *pipeline* attribute of an ETL object and the *entity_buffer* can be used for global data manipulations. It is the responsibility of the ETL object to keep track of termination condition(s) of the global program execution and not the ETLOperation objects. There are several private methods within the ETL to choose between the processing paradigms(i.e., sequential or parallel). A user of the library needs to create ETLOperation without directly interacting with it. In fact, ETLOperation is an abstract class with an unimplemented method *run(...)* as shown in the first row of table 4.4 and for this purpose, there are three main subclasses of ETLOperation: **Extractor**, **Transformer** and **Loader**. All of these classes have also pure abstract methods *extract(...)*, *transform(...)* and *load(...)* respectively shown in the same table 4.4. This is to increase modularity, maintainability, and readability of the code by making sure that any ETLOperation is either of type Extractor or Transformer or Loader. One level down, there are also two classes that inherit from both the Extractor and Loader classes: **DatabaseOps** and **FileOps**. These two classes are used to abstract away two different extraction/loading paradigms(i.e., the ones that are local to the machine and the ones that need connectivity in the network). These operations can either be used

for extraction or loading purposes with a single connection to a database or another machine and therefore, they inherit both the Extractor and Loader. The leaves of the class diagram contain the actual implementations of these methods and the user is able to initialize objects of these types.

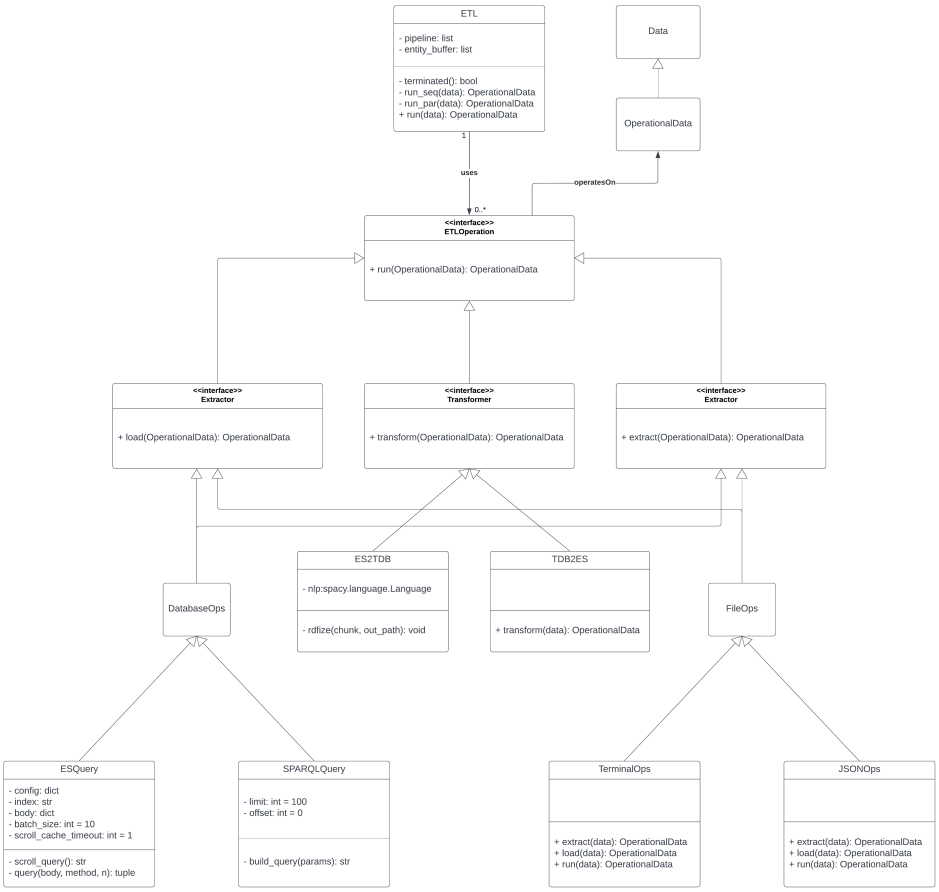


Figure 4.4: UML Class diagram

In case of any operational change in the business-logic level, one should be able to easily inherit his own *implementation class*(i.e., the one which implements the abstract method(s) of its super/parent classes) from one or more of the top-level classes[3]. For example, if the database on which the KG is usually stored has to be changed to another database system, one can either modify the existing **SPARQL-Query** class or create a new, let's say, **MongoQuery** class inherited from Extractor and Loader. An instance of MongoQuery would be as easily usable as that of SPARQLQuery. This flexibility is due to the reason that this software architecture is very

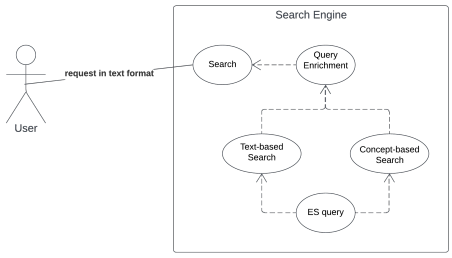
[3]The recommended classes to be inherited from are usually **DatabaseOps** and **FileOps** or **Transformer**.

similar to and almost the same as the clean software architecture described previously; moving from the top of the hierarchy to the down is analogous to moving from the most inner circle to the most outer one in the clean software architecture.

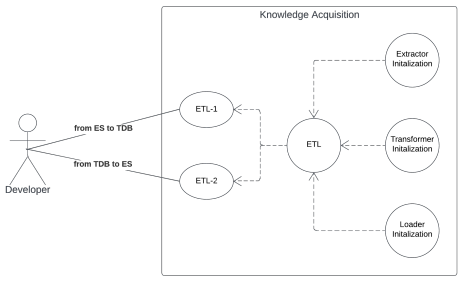
Class	Method
ETLOperation	run(data: OperationalData) → OperationalData
Extractor	extract(data: OperationalData) → OperationalData
Transformer	transform(data: OperationalData) → OperationalData
Loader	load(data: OperationalData) → OperationalData

Table 4.4: Abstract methods

Users of this project are still the developers who are going to use *knowledge acquisition* to make the resources semantically available for the final search engine processes. This being said, another type of user would be the end-users who are going to use/test the search engine by providing various textual queries and receiving the documents found and ranked by the search engine. Figure 4.5 demonstrate these use cases.



(a) Use case diagram for Knowledge Acquisition



(b) Use case diagram for Search Engine

Figure 4.5: Use case diagrams

Interaction or sequence diagram is illustrated in figure 4.6. In order to perform ETL, three operations(i.e., Extractor, Transformer and Loader) are initialized. Extractor and Loader are usually required to create a session to be able to send requests to a database. An extractor object has an attribute called *is_active* which indicates if the most recent request was able to fetch some data. If nothing was fetched then this

attribute is set to a boolean value of *false*, otherwise, it is set to *true*. The object of ETL class checks the value of *is_active* attribute of its first component(i.e., Extractor) in order to decide whether it should terminate or continue. If the termination criterion is not met then the residual components in the pipeline are executed.

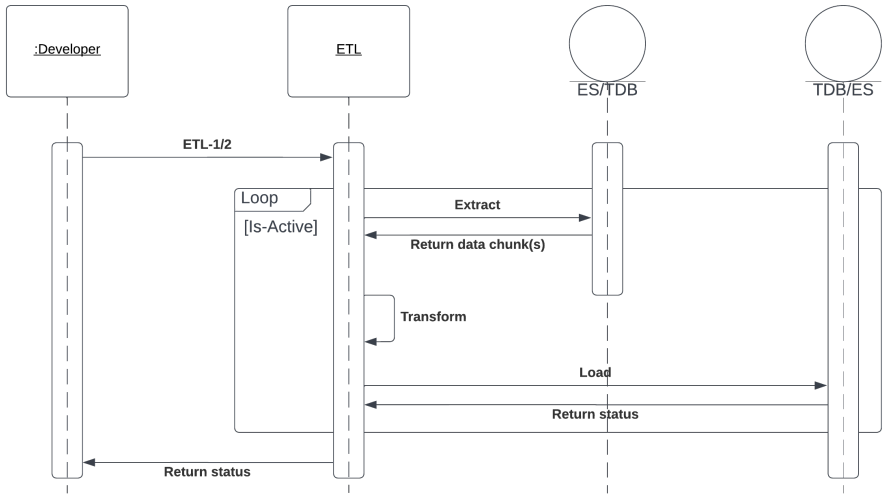


Figure 4.6: Sequence diagram

Sequential and Parallel Architecture

While developing the software to automate knowledge acquisition processes, we can also think of two main types of processing known as *sequential* and *parallel processing*. Writing a program in a sequential manner can be easy compared to the parallel one, however, there is a trade-off between the two. While sequentially running programs may usually be easier and faster to write, they may also become slower in execution time compared to that of parallel programs. Of course, this applies to programs that consists of many independent computations. The process of knowledge acquisition, in our case, contains many such computations as well. So, it is probably worth thinking about the software design from this aspect of processing too.

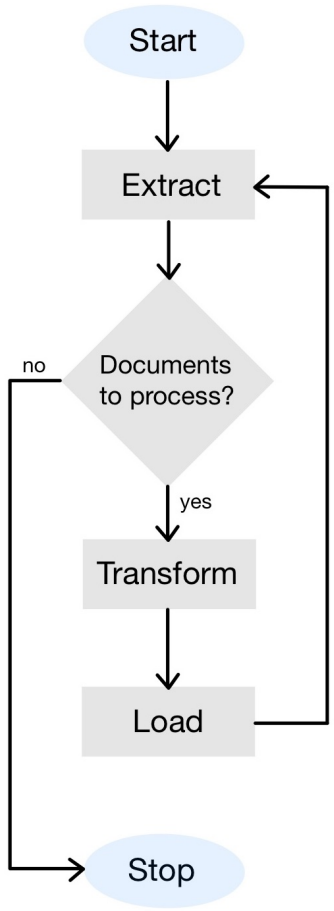


Figure 4.7: Sequential architecture

Figure 4.7 illustrate a simple sequential processing architecture that, by default, is selected when using the developed software. What this architecture tells us is that we run each subprocess (i.e., *extract*, *transform* and *load*) once per a chunk of data or documents. Note that they may be invoked more than once within the scope of the whole process and this is due to the reason that the company's database may contain hundreds of GBs, if not thousands, all of which cannot be loaded at once to the local machine or any other "regular" computer that executes the program. For this reason, the documents or the data are extracted, transformed, and loaded in as many chunks as needed. If one iteration of ETL can only deal with 1000 documents and there are 10^9 documents in the database, a million ETL processes are going to be executed one after another, in a sequential manner, in order to acquire 100% of the knowledge completely.

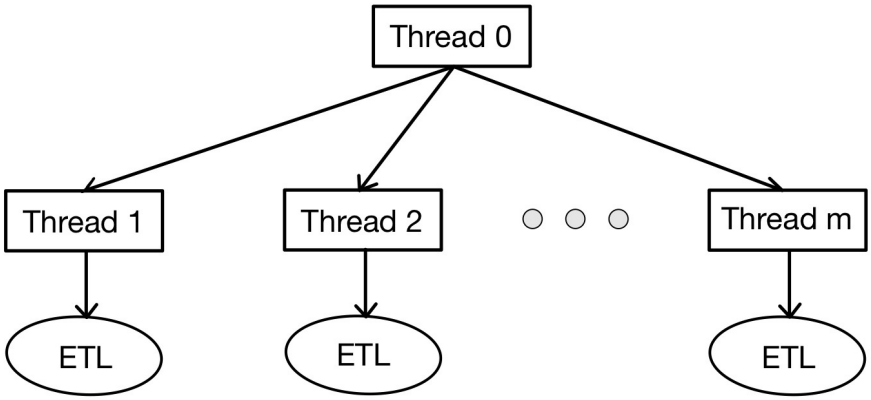


Figure 4.8: Parallel architecture

In contrast to the sequential processing, it is also possible for one to separate ETL process to not wait for the other since they are completely independent. Realization of this simple fact leads us to develop a simple, yet faster-running, parallel processing architecture shown in figure 4.8.

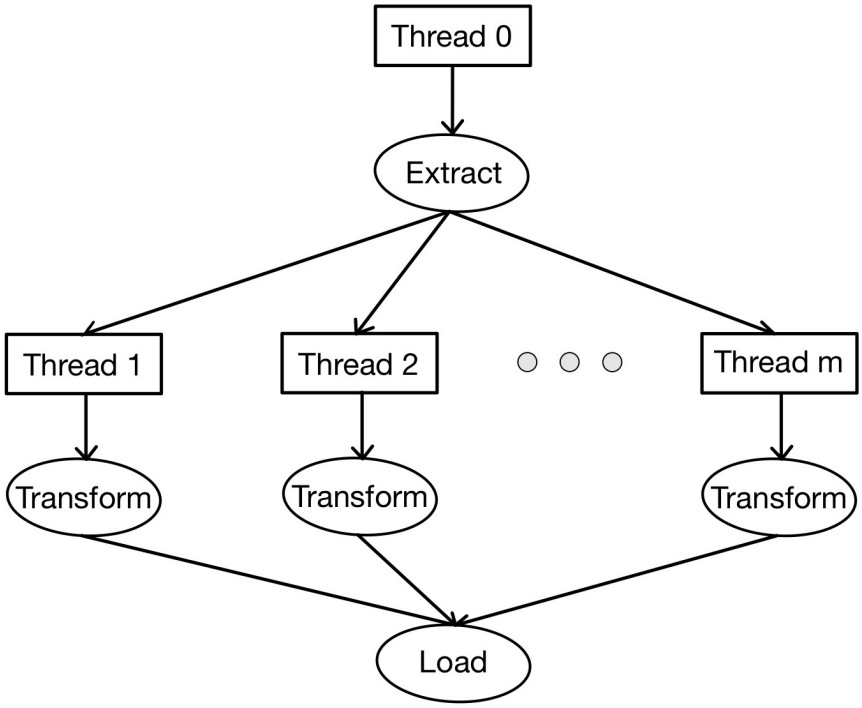


Figure 4.9: Improved parallel architecture

Enhanced version of the architecture illustrated in figure 4.8 can be easily realized if we excluded *extract* and *load* operations from running in a parallel fashion and

only kept the *transform* operation. The idea behind this architecture shown in figure 4.9 is the fact that each extraction and loading operation requires sending/receiving requests from/to a database running on some external network. It may be time-consuming to send and receive packets through the network when these operations are repeated many times. This is the reason, we can also improve the first parallel architecture by bringing the parallelism only to the *transformation layer*. Therefore, instead of many extract-transform-load operations, we perform mass extraction then transformation of the vast amount of fetched documents in parallel, and finally, mass loading into another database.

4.2.4 Specification and Implementation

In this section, the developed software is described in terms of core implementational details that help to visualize and understand the underlying working principles of the carried operations. Mainly, there are several (sub)processes that have been described:

1. Extract-Transform-Load pipeline execution
2. Extraction process
3. Transformation process
4. Loading process

The subprocesses(i.e., Extraction, Transformation and Loading) are first described in terms of their roles in the whole pipeline, and then the following sections break down these components by providing pseudo-codes and additional comments.

ETL Pipeline

ETL pipeline is a sequence of **ETL Operations** that are executed in the given order. The existence of such a pipeline is for better modularity and readability. Components can easily be added, modified or removed from the pipeline. There are three

ETL Operations or pipeline components that can be added to the ETL pipeline: *Extractor*, *Transformer* and *Loader*. Figure 4.10 illustrates an example pipeline with these components and * sign on the top right side of each component indicates that there can be 0 or more such operations. The order of the components can also be arbitrary, however, it defines the processing order accordingly and therefore, depends on the use case of a user. Despite that, the common order of components in the pipeline is as shown below. Each operation takes in an input and returns an output of **Data** type. This constraint on the pipeline components is very helpful to build robustness and modularity.

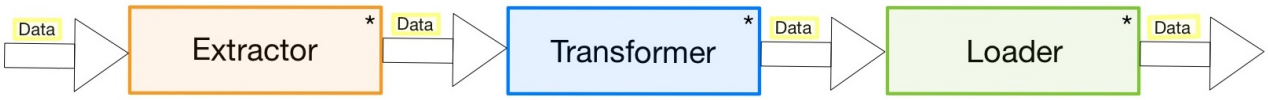


Figure 4.10: ETL pipeline

The **Data** type is a simple class that has been created to serve as a unified data storage. By doing so, we can have restrictions, constraints and processes to check the validity of the data, if needed. Table 4.5 shows the attributes and methods of this class.

Attribute	**Description**
data	Can be used to hold any type of data, however, the recommended format is the dictionary
Method	**Description**
-	-

Table 4.5: Data Class Attributes/Methods

There are two different ETL pipelines used for the knowledge acquisition named as *ETL-1* and *ETL-2*. ETL-1 is to build and store a knowledge graph (KG) in a TDB database by fetching the raw documents from the original ES database. ETL-2 is to

build a semantic ES (S-ES) database by using the already existing KG and make the documents searchable in a more semantic manner. For the detailed explaination of how the pipeline works, I am going to give demonstrative examples for both ETL pipelines and describe them step-by-step.

ETL-1 pipeline contains **ESQuery** as an *extractor*, **ES2TDB** as a *transformer* and **TerminalOps** as a *loader*. The execution of this pipeline yields KG built upon the documents stored the original ES database. First, ESQuery fetches the documents chunk-by-chunk with the given document count per chunk and the total chunk count. Figure 4.11a illustrates an example row of the ES database. Fetched documents are transformed by ES2TDB into OTTR instances as shown in figure 4.11b. This KG is then stored in a Jena's TDB database by the TerminalOps. Some of the final RDF/Turtle triples represented in the store KG are illustrated by figure 4.11c (see figure A.3 for the full version).

ETL-2 pipeline contains **SPARQLQuery** as an *extractor*, **TDB2ES** as a *transformer* and **JSONOps** as a *loader*. This pipeline is executed after the execution of the first one and therefore, the existence of the KG as the single source of truth is already guaranteed. The necessary instances from the KG stored in the TDB database are fetched through one or more SPARQL queries as shown in figures 4.12a and 4.12b[4] [5] [6]. Obtained responses are transformed to a suitable dictionary format according to the already-specified schema of the S-ES database. After the transformation of the fetched instances, the prepared Python dictionary is dumped to a JSON file as the result of the JSONOps execution. Figure 4.12c illustrates the transformed version of the entities in a JSON format.

[4] tp = <https://ontologies.traceparts.com/>

[5] gist = <https://ontologies.semanticarts.com/gist/>

[6] rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PartFamily	PartNumber	PartFamilyName	PartNumberName
90-22112018-056525	LC1BL34U31	[en] TeSys B contactor 4P 800A 240V AC - LC1BL34U31	[en] TeSys B contactor 4P 800A 240V AC - LC1BL34U31

(a) Document fetched from the ES database

tp:createPartNumberInstance(tp:PNLC1BL34U31, tp:IDLC1BL34U31, tp:PF90-22112018-056525, tp:Text-6378022537900392513).

tp:createPartFamilyInstance(tp:PF90-22112018-056525, tp:ID90-22112018-056525, tp:Text-2992260873777870348).

tp:createTextInstance(tp:Text-6378022537900392513, ”TeSys B contactor 4P 800A 240V AC”^^xsd:string, tp:Language-def, (tp:ID4P,tp:ID800A,tp:ID240V)).

tp:createTextInstance(tp:Text-2992260873777870348, ”TeSys B contactor 4P 800A 240V AC - LC1BL34U31”^^xsd:string, tp:Language-def, (tp:ID4P,tp:ID800A,tp:ID240V,tp:IDLC1BL34U31)).

tp:createIdInstance(tp:IDLC1BL34U31, ”LC1BL34U31”^^xsd:string).
tp:createIdInstance(tp:ID90-22112018-056525, ”90-22112018-056525”^^xsd:string).
tp:createIdInstance(tp:ID4P, ”4P”^^xsd:string).
tp:createIdInstance(tp:ID800A, ”800A”^^xsd:string).
tp:createIdInstance(tp:ID240V, ”240V”^^xsd:string).

(b) OTTR instance

Subject	Predicate	Object
tp:PNLC1BL34U31	rdf:type gist:isIdentifiedBy gist:isDescribedIn tp:hasPartFamily	tp:PartNumber tp:IDLC1BL34U31 tp:Text-6378022537900392513 tp:PF90-22112018-056525
tp:PF90-22112018-056525	rdf:type gist:isIdentifiedBy gist:isDescribedIn	tp:PartFamily tp:ID90-22112018-056525 tp:Text-2992260873777870348
tp:Text-6378022537900392513	rdf:type gist:containedText gist:isExpressedIn	gist:Text ”TeSys B contactor 4P 800A 240V AC - LC1BL34U31” tp:Language-def
tp:IDLC1BL34U31	rdf:type gist:uniqueText gist:isPartOf	gist:ID ”LC1BL34U31” tp:Text-6378022537900392513

(c) RDF/Turtle triples

Figure 4.11: ETL from ES to TDB example

```
select ?subject ?predicate ?object
where {
  ?subject ?predicate ?object
}
limit 5
offset 0
```

(a) SPARQL query

Subject	**Predicate**	**Object**
tp:ID10-12032019-066017	rdf:type	gist:ID
tp:ID10-12032019-066017	gist:uniqueText	”10-12032019-066017”
tp:IDZB5FA46C0	rdf:type	gist:ID
tp:IDZB5FA46C0	gist:uniqueText	”ZB5FA46C0”
tp:Text79763625448103550	rdf:type	gist:Text

(b) SPARQL response (KG)

Key	**Value**
doc_id	90-22112018-056525:LC1BL34Q31
uri	https://ontologies.traceparts.com/PF90-22112018-056525
concept_uris	tp:ID4P tp:IDLC1BL34Q31 tp:ID800A tp:ID240V
searcable_texts	def: TeSys B contactor 4P 800A 240V AC - LC1BL34Q31 en: TeSys B contactor 4P 800A 240V AC - LC1BL34Q31

(c) RDF/Turtle triple

Figure 4.12: ETL from TDB to ES example

ETL algorithm shown in 1 executes each pipeline component(i.e., extractor, transformer and loader) in the given order. If a user provides {"parallelism": 0} in the data object to be given to *run* method of the ETL object, the components are executed in a sequential fashion, one by one. Otherwise, the transformation operation is executed in a parallel fashion.

Algorithm 1 ETL

Require: `OperationalData`

Ensure: `OperationalData`

```
while the process should not terminate do
    if sequential run then
        for component in pipeline do
            data = component.run(data)
        end for
    else                                                    ▷ for parallel run
        get extractor, transformer, loader from the pipeline
        extractor.run(data)
        allocate m buffers inside data object
        allocate m threads to run transformer on the respective buffer
        for thread in threads do
            thread.run()                ▷ Each thread performs transformations on a subset of documents and stores the result temporarility on the corresponding index in buffer
        end for
        wait for all m threads to finish
        data = loader.run(data)
    end if
end while
```

The *pipeline* attribute of an ETL object can be initialized with an arbitrary number of ETL operations and in an arbitrary order. Such an arbitrary pipeline would work just fine if executed sequentially. Parallelized execution is only suitable for this project since there is only one of each operation per the whole ETL process. If the pipeline was to contain several extractors/loaders then a more generic implementation would be required to optimize it.

Extractor

The Extractor class is a process-only class[7]. Table 4.6 shows the attributes and methods of the Extractor class. There are no attributes and only two abstract methods one of which(i.e., `run(...)`) is inheretied from the parent class ETLOperation.

Attribute	**Description**
-	-
Method	**Description**
extract(data: OperationalData) → OperationalData	abstract
run(data: OperationalData) → OperationalData	abstract, inherited

Table 4.6: Extractor Class Attributes/Methods

Algorithm 2 ESQuery/SPARQLQuery-Extractor

Require: requestBody, method, url, numChunk
Ensure: docs (as a list of numChunk chunks)
  session = createSession() ▷ created once within an object
  if scrollRequest **then** ▷ getting response partially
    requestBody = buildScrollQuery() ▷ different implementation for different databases
  end if

[7]By process-only class, I mean a class that consists of only methods and not attributes since the whole purpose of using them is mostly processing the given data and returning the modified version of it back.

response = session.request(requestBody, method, url)
extract **header** and **hits** from **response**
chunks = [] ▷ initialize chunks as an empty list
chunkSize = len(hits) / numChunk
for i in $\{0, \cdots, \text{numChunk} - 1\}$ **do**
    chunks.append(hits[$i * \text{chunkSize} : (i+1) * \text{chunkSize}$) ▷ add sublist of documents
end for
chunks.append(hits[$(\text{numChunk} - 1) * \text{chunkSize} :$]) ▷ add the residual documents, if any
return header, chunks

Algorithm 2 illustrates how the specific extraction processes from ElasticSearch (ESQuery) and TDB(SPARQLQuery) are performed. The only important difference between these processes is in the query-building part. Scroll queries are known as queries performed on the databases resulting in partial responses. When the response may be too huge in size, we request only a part of it iteratively until we reach to the end of it and this is the main reason why scroll queries are very useful. More specifically, the scroll queries for the request body are built differently for ES and TDB endpoints. For example, building a scroll query requires a *scroll ID* of the last query in the ES database while there is an offset value set by the user to get the next piece of partial response in the TDB database. So, the implementation of building scroll queries has solely built upon scroll IDs and offset values for the respective endpoints.

Transformer

Transformer is a pure abstract class which is used to define an interface for different transformation implementations. Table 4.7 illustrates the attributes and methods of this class.

Attribute	**Description**
nlp	spacy.language.Language
Method	**Description**
transform(data: OperationalData) → OperationalData	abstract
run(data: OperationalData) → OperationalData	abstract, inherited

Table 4.7: Transformer Class Attributes/Methods

Algorithms 3 and 4 illustrate how the specific transformation processes are executed to transform entites coming from ES(original database) to TDB(knowledge graph storage) and from TDB(knowledge graph storage) to ES(semantic ES database) respectively.

Algorithm 3 ES2TDB-Transformer

Require: docs

Ensure: rdf/ttl file

```
file = open(filepath)
for doc in docs do
    fetch pfID from doc
    fetch pnNumber from doc
    fetch pfNamesDict from doc
    fetch pnNamesDict from doc
    file.write(processID(pfID))        ▷ processID returns ID intialization string in OTTR format
    file.write(processID(pnNumber))
    for language, pfName in pfNamesDict do
        file.write(processPF(nlp, pfID, pfName, language)    ▷ processPF returns PartFamily initialization string in OTTR format
    end for
    for language, pfName in pnNamesDict do
        file.write(processPN(nlp, pnNumber, pnName, language)    ▷ processPN returns PartNumber initialization string in OTTR format
    end for
end for
compileOTTR(filepath)        ▷ compilation of generated OTTR instance file(s)
```

As shown in algorithm 3, the transformer *ES2TDB* creates an empty OTTR instance file first. Each document given as the argument to this function is iterated one-by-one and part family number, part name number, part family description and part number description are gathered and appended to the instance file. However, gathering IDs for part families/numbers requires NLP[8] to extract them from the texts.

[8]Natural Language Processing

For this purpose, we use spaCy[9] library which is well-known and used reliably by many other companies. To extract IDs from the given text, a regular expression is used as a heuristic. Finally, the instance file is compiled to RDF/Turtle triples with *Lutra* which is a compiler for OTTR language.

Algorithm 4 TDB2ES-Transformer

Require: docs, responses

Ensure: json text

```
for response in responses do
    result =                                          ▷ initialize empty dictionary
    docID = response['pf_text'] + response['pn_text']
    if docID has not been processed already then
        docs.append(result)                           ▷ mark it as processed
    else
        result = documentWith(docID)                  ▷ find and return the corresponding
document
    end if
    result['docID'] = docID                           ▷ the first field of final json
    result['uri'] = response['pf_uri']                ▷ the second field of final json
    if result has no field named 'concept_uris' then
        result['concept_uris'] = []                   ▷ initialize concept uris as an empty list
    end if
    fidURI = response['fid_uri']                      ▷ PartFamily uri
    nidURI = response['fid_uri']                      ▷ PartNumber uri
```

[9] https://spacy.io

if fidURI is not in result['concept_uris'] **then**

result['concept_uris'].append(fidURI)

end if

if nidURI is not in result['concept_uris'] **then**

result['concept_uris'].append(nidURI)

end if

if result has no field named 'searchable_texts' **then**

result['searchable_texts] = ▷ initialize searchable texts as an empty dictionary

end if

flang = response['flang_uri'] ▷ uri of the language of the PartFamily text/description

nlang = response['nlang_uri'] ▷ uri of the langauge of the PartNumber text/description

result['searchable_texts'][flang] = response['ftext'] ▷ PartFamily text/description result['searchable_texts'][nlang] = response['ntext'] ▷ PartNumber text/description

end for

The second transformer *TDB2ES* as shown in algorithm 4 is used to transform relevant subgraphs in the previously built KG to JSON format with desired fields. Since we think that storing each document represented as several fields such as document ID, part family URI, candidate concepts URIs and searchable texts relevant to that document is useful, the transformer sends a SPARQL query to the relevant Jena TDB endpoint and gathers the mentioned fields and relevant in a dictionary format.

Loader

Loader class is used to abstract away the operations copying/moving a chunk of data from the current path to the target environment(i.e., another path on a local

machine or a database host on another network). Table 4.8 illustrates the attributes and methods of this class.

Attribute	**Description**
-	-
Method	**Description**
load(data: OperationalData) → OperationalData	abstract

Table 4.8: Loader Class Attributes/Methods

Algorithm 5 illustrates how the specific loading process is executed in this project. Currently, loading triples to the Jena TDB database requires copying the Turtle files to the directory *staging/* of the database running on Docker. This directory is shared with the local machine. However, it is recommended to avoid such a loading process in the future since it is not elegant and maintainable.

Algorithm 5 TerminalOps-Loader

Require: source_directory, target_directory
Ensure: fuseki database is updated
copy *.ttl from the **source_directory** to the **target_directory** ▷ target is fuseki /staging/ directory
execute **tdbloader** to load from */staging/*.ttl* to *fuseki/databases/responding* database

The second loading process which takes care of transformed Turtle triples is demonstrated in algorithm 6. Currently, obtained dictionary from the previous transformation(see algorithm 4) are stored as a JSON file on the local machine. This file can be loaded into ES database afterwards.

Algorithm 6 JSONOps-Loader

Require: docs, target_file
Ensure: json file has been produced for S-ES database
dump all the **docs** in a json format and save it in a **target_file**

4.2.5 Testing

Explicitly testing software is an effective way of finding potential bugs hidden in the code. There are two main types of testing: *functional* and *non-functional* testing. Functional testing is to assure that the developed piece of code within the software works as expected according to the software specification or UML use case diagrams. As an example, *unit testing*, *integration testing*, *system testing*, etc. can be given. On the other hand, non-functional testing focuses on the operational aspects. *Performance testing*, *security testing*, *usability testing*, etc. can be given as an example of non-functional testing.

Unit testing is used to test the individual components of the developed software. Such type of tests is usually conducted by the software developers. It is very effective to write unit tests (even sometimes beforehand) since they focus on each component of the software individually which makes future debugging very easy for the developer. Several unit tests have been written to test knowledge acquisition components in this project as well.

The most error-prone components of our software have been heuristics used for concept extraction. Since the process of acquiring knowledge from SMEs directly is not maintainable, we developed heuristics to extract some of the concepts from raw text automatically. For example, IDs of CAD[10] models have been extracted in this way. Such heuristics usually describe the format of the tokens such as what characters are allowed to be used and how many of them, how they can arrange a sequence, and so on. Improving heuristics is an iterative process of trial and error since there is no direct knowledge source(i.e., a human expert).

[10]Computer Aided Design

5 Results and Discussion

5.1 Results

During the internship, I have been able to understand Knowledge Acquisition, its role in improving a search engine, its pipeline components and to propose software architecture, different processing paradigms, and algorithms to implement them. Acquiring knowledge from a relational database of the company is the first process of IR improvement. It is also worth to mention that the software that I have been developing is going to be used once since all it does is to take the original database of the company and build two databases(i.e., one for storing KG and the other for semantic product descriptions).

5.1.1 Library

As a result of different kinds of processes required for the knowledge acquisition, I have built a library to abstract away most of the low-level implementation details. One of the most important demands for this project is maintainability and readability. To achieve this goal, I have mainly focused on the clean software architecture as mentioned in the previous chapter. The library helps us to develop any high-level software application(s) on top of it. Since *Python3+* has been chosen to be the main programming language, the library could be *import*ed into a project to be used.

Library	Used for
`spacy`	NLP on user queries/model descriptions
`requests`	sending requests to databases
`threading`	parallization of ETL operations
`abc`	making top-level classes abstract and reinforcing developers to implement abstract methods
`json`	json file operations
`logging`	logging
`configparser`	parsing ".ini" configuration files
`urllib.parse`	encoding IDs for OTTR

Table 5.1: External libraries

5.1.2 Command-Line Interface

Several external libraries have been used in this project for different purposes. Two most important ones are *spacy* and *requests*. Instead of using the functionalities provided by these libraries directly, I have developed seperate classes that get rid of unnecessary and irrelevant functionalities and keep only the ones that are relevant to this project.

Spacy is a well-designed library used in industry for natural language processing. However, it has many functionalities that would be considered overhead for this project. Due to this reason, I have built a custom language model that is suitable for our needs.

Requests library helps developers to send/recieve information to/from endpoints. The send a request, request body, target url, method(i.e., GET, POST, etc.) have to be provided. Building scroll queries as described previously is necessary for abstraction of writing request bodies and/or target urls manually.

By using the devloped library and custom language model (see appendix A.4.1),

a simple CLI[1] has been developed. Both ETL-1 and ETL-2 processes can be run by the same program. All a user is required to do is to enter his/her choice of process after the prompt.

5.1.3 Tests

Invalid Configuration Handling

To execute ETL operations, one needs to provide the program a configuration. This configuration is recommended to be in the ".ini" file format, however, several environment variables can also be used. An example configuration file looks like in figure 5.1. It is parsed and used by different operations such as Extractor and Loader (i.e., ESQuery).

```
[DEFAULT]
data_path = ../data/sample-data-test-IDs.csv
ottr_lib_path = ottr/lib.stottr
rdf_out_dir = ottr/
lutra_path = ~/Downloads/lutra.jar

# number of threads in ETL process
n_thread = 4
# number of responses(documents) processed by a thread
n_chunk_per_thread = 25

[ES]
dstro_es_url = http://localhost
dstro_es_port = :9200
dstro_es_port2 = :9300
dstro_es_product_idx = es_apollo_test

[ES_NEW]
es_url = http://localhost
dstro_es_port = :0000

[JENA]
tdb_url = http://localhost
dstro_es_port = :0000
```

Figure 5.1: Configuration File example

The first four attributes can also be set with the help of environment variables. If neither environment variables are set nor configuration file misses some of the necessary attributes, program will halt with a self-descriptive error message.

Invalid Document Handling

As the extraction process of IDs have been implemented with *regex*[2] there are unittests to test edge cases. There are two cases: extracted ID is not a real ID or there is an ID which is not extracted. It is good to develop tests for each case.

[1] Command-Line Interface
[2] Regular Expression

String	Is ID?
10-23GGEZ-SHEESH	Yes
240V	Yes
10-23ggez-sheesh	No
800A	Yes
LCDAMOGUSRT34	Yes

Table 5.2: Examples for ID verification

As shown in table 5.2, the first and the last strings are actual IDs whereas the third one is not. However, the second and the fourth strings are not real IDs either although our heuristic currently cannot classify such types correctly.

5.2 Discussion

We can see that the developed software may still contain some bugs and terminate unexpectedly. This is due to several factors such as lack of unittests and edge cases given to the existing tests, poorly designed production and consumption of OperationalData objects, and imperfect heuristic implementation. In fact, the library and the software need to be improved furthermore. Several factors can be improved as shown below:

- Heurictics can be added or modified according to SMEs knowledge
- More test cases for individual ETL operations can be added or even tests can be automated
- Data production and consumption can be handled with encapsulated furthermore
- Configuration file template can be structured more clearly
- Benchmarks can be designed to test the performance of each operation as well as the whole to choose the best between different processing architectures

- etc.

We hope that the current software architecture, some core ideas about the pipeline components and processing paradigms and algorithms for them are not very far away from multiple future versions of the library and the software. Several modifications have to be made only on the implementational level and not on the architectural level.

6 Conclusion

6.1 On the use of ontologies

Ontologies are used in different applications with mainly the unified goal of having a shared vocabulary with agreed-upon semantics. Such applications have emerged from domains such as fraud detection, web mining, search engines, legacy system integration, e-learning, data-level data integration, semantic publishing, and so on [22, 25]. You can imagine how having a shared vocabulary could potentially help to create more easily integratable systems and therefore, an ecosystem. Ontologies are very useful for such purposes.

Although developing and using ontologies can be very helpful for many real-world applications, they can also be painful to deal with in other areas of interest. The first problem is that it is not probably practically possible to define everything very rigorously by using some subset of the first-order logic. For example, how could one try to solve an image classification problem with the help of an ontology, or play chess, or solve a maze? Such problems require different solutions than using pure ontologies. The second issue with using ontologies is that they have their own limitations in terms of the expressivity. To make the inference process tractable and decidable, there are different types of *profiles* that are suitable for different use cases. The trade-off is obvious - more expressivity, less tractablity. For example, if you worked with the RDF framework, it would be impossible for you to express "a class being a subclass of some another class" since there is no such object property defined, and to be able to do it, RDFS would be one of the options to choose. However, as mentioned earlier, every framework has the trade-off between expressivity and

tractability.

6.2 On the use of OTTR

The idealogy behind OTTR is simple and powerful - *using templates to build knowledge bases with better readability, maintainability and usability*. What it gives to its users is a more efficient process of dealing with ontologies and knowledge bases. Templates used in OTTR language prevent many potential repetitions by making the whole process more modular. Such separation between the design and the content allows one to develop a template library once and use it as much as needed to add new instances to the knowledge base.

Serialization of OTTR templates into RDF/OWL format is another benefit of using them. Imagine you have built a template library that can be used to add instances to the KB and the library itself could be loaded to the same KB. You would not need to store the library file separately on your local machine or some separate database than the one which contains the instances. So, the library file can be fetched from the same database either to check the validity of the triples or add new instances more reliably and easily.

6.3 Experiences gained during the internship

To be able to solve the knowledge acquisition task, we encountered many problems along the way. These problems included thinking individually and brainstorming as a team with my supervisor. Communication and shared vocabulary played an important role during such times since when having a discussion about ontologies, people can easily misunderstand each other due to ambiguously defined and/or used words. However, because of such situations the need for rigor became very obvious to me. I understood the non-triviality of building a good ontology, building trustfulness by centeralization[1], decreasing performance issues when scaling up the database size, etc. These real-world problems made me realize the importance

[1] KG has been built to play the role of single unified information source.

of scientific methods, differences between theory(thinking without constraints) and practice(engineering - thinking with constraints), communication and organization.

Bibliography

[1] Wikipedia contributors. *Search engine results page — Wikipedia, The Free Encyclopedia*. [Online; accessed 21-June-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Search_engine_results_page&oldid=1094572512.

[2] Wikipedia contributors. *Web crawler — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-June-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Web_crawler&oldid=1082281684.

[3] Wikipedia contributors. *Search engine indexing — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-June-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Search_engine_indexing&oldid=1088909238.

[4] Wikipedia contributors. *Web query — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-June-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Web_query&oldid=1088145469.

[5] Wikipedia contributors. *Proximity search (text) — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-June-2022]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Proximity_search_(text)%5C&oldid=1058506896.

[6] Wikipedia contributors. *Concept search — Wikipedia, The Free Encyclopedia*. [Online; accessed 14-June-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Concept_search&oldid=1093304368.

[7] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. *Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models*. 2017. DOI: 10.48550/ARXIV.1708.08296. URL: https://arxiv.org/abs/1708.08296.

[8] Nikolaos Trokanas, Linsey Koo, and Franjo Cecelja. "Towards a Methodology for Reusable Ontology Engineering: Application to the Process Engineering Domain". In: Computer Aided Chemical Engineering 43 (2018). Ed. by Anton Friedl et al., pp. 471–476. ISSN: 1570-7946. DOI: https://doi.org/10.1016/B978-0-444-64235-6.50084-X. URL: https://www.sciencedirect.com/science/article/pii/B978044464235650084X.

[9] Wikipedia contributors. *Bayes' theorem — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-June-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Bayes%27_theorem&oldid=1094315386.

[10] Wikipedia contributors. *Search engine optimization — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Search_engine_optimization&oldid=1082522144. [Online; accessed 15-June-2022]. 2022.

[11] Lien-Fu Lai et al. "Developing a fuzzy search engine based on fuzzy ontology and semantic search". In: *2011 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2011)*. 2011, pp. 2684–2689. DOI: 10.1109/FUZZY.2011.6007378.

[12] Wikipedia contributors. *Tf–idf — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Tf%E2%80%93idf&oldid=1092578440. [Online; accessed 10-June-2022]. 2022.

[13] Wikipedia contributors. *PageRank — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=PageRank&oldid=1094205204. [Online; accessed 12-June-2022]. 2022.

[14] Dario Bonino et al. “Ontology driven semantic search”. In: *WSEAS Transaction on Information Science and Application* 1.6 (2004), pp. 1597–1605.

[15] Tania Tudorache et al. “WebProtégé: A collaborative ontology editor and knowledge acquisition tool for the web”. In: *Semantic web* 4.1 (2013), pp. 89–99.

[16] Melvin Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[17] Adam Pease and Geoff Sutcliffe. “First Order Reasoning on a Large Ontology.” In: *ESARLT* 257 (2007).

[18] Jean-Pierre Laurent. “Proposals for a Valid Terminology in KBS Validation”. In: *Proceedings of the 10th European Conference on Artificial Intelligence*. ECAI ’92. Vienna, Austria: John Wiley & Sons, Inc., 1992, pp. 829–834. ISBN: 0471936081.

[19] Mieczyslaw L. Owoc, Malgorzata Ochmanska, and Tomasz Gladysz. “On Principles of Knowledge Validation”. In: *Validation and Verification of Knowledge Based Systems: Theory, Tools and Practice*. Ed. by Anca Vermesan and Frans Coenen. Boston, MA: Springer US, 1999, pp. 25–35. ISBN: 978-1-4757-6916-6. DOI: 10.1007/978-1-4757-6916-6_2. URL: https://doi.org/10.1007/978-1-4757-6916-6_2.

[20] Wikipedia contributors. *Ontology (information science) — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-June-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Ontology_(information_science)%5C&oldid=1085831526.

[21] Wikipedia contributors. *Line (geometry) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Line_(geometry)&oldid=1092767429. [Online; accessed 22-June-2022]. 2022.

[22] University of Cape Town C. Maria Keet. *An Introduction To Ontology Engineering*. Maria Keet, 2018. URL: https://people.cs.uct.ac.za/~mkeet/files/OEbook.

[23] *gist*. 2022. URL: https://www.semanticarts.com/gist/.

[24] *The Clean Architecture*. 2012. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[25] *What are ontologies?* 2022. URL: https://www.ontotext.com/knowledgehub/fundamentals/what-are-ontologies/.

A Appendices

A.1 Gist

There are much more concepts than shown below in the original gist upper ontology, however, the figure is to illustrate some of the useful concepts that have been used in the project. More specifically, concepts of **ID**, **Text**, **Language** and relations of *is expressed in*, *is part of*, *unique text*, *contained text* have been integrated into the project.

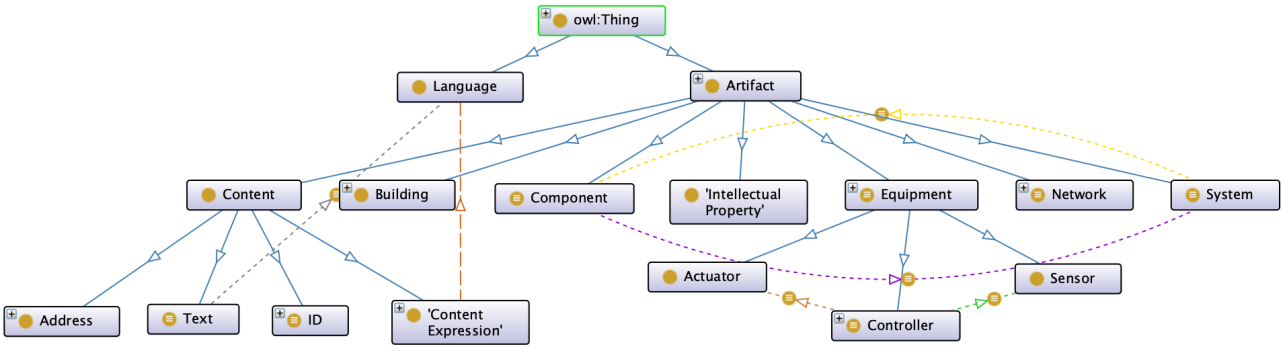


Figure A.1: gist view

A.2 OTTR

```
16 tp:isPartOf[gist:ID ?id, gist:Text ?text] :: {
17     ottr:Triple(?id, gist:isPartOf, ?text)
18 }.
19
20 tp:createLanguageInstance[tp:Language ?language, xsd:string ?language_str] :: {
21     ottr:Triple(?language, rdf:type, tp:Language),
22     ottr:Triple(?language, tp:hasName, ?language_str)
23 }.
24
25 tp:createTextInstance[gist:Text ?text, xsd:string ?text_str, gist:Language ?language, List<gist:ID> ?ids] :: {
26     ottr:Triple(?text, rdf:type, gist:Text),
27     ottr:Triple(?text, gist:isExpressedIn, ?language),
28     ottr:Triple(?text, gist:containedText, ?text_str),
29     cross | tp:isPartOf(++?ids, ?text)
30 }.
```

Figure A.2: OTTR library example

A.3 ETL

Subject	**Predicate**	**Object**
tp:PNLC1BL34U31	rdf:type gist:isIdentifiedBy gist:isDescribedIn tp:hasPartFamily	tp:PartNumber tp:IDLC1BL34U31 tp:Text-6378022537900392513 tp:PF90-22112018-056525
tp:PF90-22112018-056525	rdf:type gist:isIdentifiedBy gist:isDescribedIn	tp:PartFamily tp:ID90-22112018-056525 tp:Text-2992260873777870348
tp:Text-6378022537900392513	rdf:type gist:containedText gist:isExpressedIn	gist:Text ”TeSys B contactor 4P 800A 240V AC - LC1BL34U31” tp:Language-def

tp:Text-2992260873777870348	rdf:type gist:containedText gist:isExpressedIn	gist:Text ”TeSys B contactor 4P 800A 240V AC” tp:Language-def
tp:IDLC1BL34U31	rdf:type gist:uniqueText gist:isPartOf	gist:ID ”LC1BL34U31” tp:Text-6378022537900392513
tp:ID90-22112018-056525	rdf:type gist:uniqueText gist:isPartOf	gist:ID ”90-22112018-056525” tp:Text-2992260873777870348
tp:ID4P	rdf:type gist:uniqueText gist:isPartOf gist:isPartOf	gist:ID ”4P” tp:Text-6378022537900392513 tp:Text-2992260873777870348
tp:ID800A	rdf:type gist:uniqueText gist:isPartOf gist:isPartOf	gist:ID ”800A” tp:Text-6378022537900392513 tp:Text-2992260873777870348
tp:ID240V	rdf:type gist:uniqueText gist:isPartOf gist:isPartOf	gist:ID ”240V” tp:Text-6378022537900392513 tp:Text-2992260873777870348

A.4 Responding Knowledge Acquisition Software

A.4.1 Library usage

```
from etl import *


class MyExtractor(Extractor):
        # your implementation

class MyTransformer(Transformer):
        # your implementation

class MyLoader(Loader):
        # your implementation


if __name__ == "__main__":
        # declaring an Extractor object
        extractor = MyExtractor(None)

        # declaring a Transformer object
        transformer = MyTransformer(None)

        # declaring a Loader object
        loader = MyLoader(None)

        # initializing OperationalData object
        data = OperationalData({})
```

```
# initalizing ETL object
etl = ETL(pipeline=[extractor, transformer, loader])

# executing the ETL pipeline processes
etl.run(data)
```