Zurich University
of Applied Sciences

School of
Engineering

InIT Institute of Applied
Information Technology

Bachelor thesis

Information retrieval with context-recall in human-computer conversations

Author Matteo Togni
Sandro Panighetti

Main supervisor Mark Cieliebak

Date 08.06.2017

Zurich University
of Applied Sciences

School of Engineering

DECLARATION OF ORIGINALITY

Bachelor's Thesis at the School of Engineering

By submitting this Bachelor's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Bachelor thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City, Date: Signature:

....

.....

.....

The original signed and dated document (no copies) must be included after the title sheet in the ZHAW version of all Bachelor thesis submitted.

Zurich University
of Applied Sciences

Abstract

In today's world, messaging applications are becoming more and more popular, especially with the advent of smart-phones. Chatbots, a software to automate answers, are often built upon these apps.

In this thesis, we focus on building a chatbot capable of answering questions about film productions by retrieving information from *The Internet Movie Database*. We investigate the key elements required to build a chatbot and examine different context-recalling approaches to build upon rule-based solutions.

Retrieving relevant information from the relational database of IMDB is a challenging mission, as it involves mapping natural language onto a SQL query. Thus, the relational database is transformed into a RDF store, a graph database which stores semantic facts. In doing so, data is represented in a more similar form to natural language, and is therefore easier to query.

Furthermore, a context-recalling solution involving the previously migrated RDF store is proposed. The metadata in the semantic database is used to relate new sentence items to the current context.

The combination of these two solutions permits the generation of more valuable queries.

Due to time constraints and the unavailability of a suitable comparative dataset, a thorough evaluation of the proposed solution could not be conducted. Nevertheless, it was possible to determine that using a specific domain of data in form of triples simplifies the understanding of words. Thus, facilitating information retrieval and context-recalling approaches. An additional insight gained, was that evaluating complex relationships in graphs can be computationally expensive.

Further development of the prototype would be necessary in order to achieve a workable solution.

Zusammenfassung

Heutzutage sind Instant-Messaging-Anwendungen insbesondere durch die Einführung von Smartphones immer populärer. Chatbots, Softwares, die Antworten automatisch erstellen, sind oft auf solchen Applikationen aufgebaut.

In dieser Arbeit, liegt der Fokus darauf, einen Chatbots aufzubauen, der die Fähigkeit hat, Fragen über Filme zu beantworten. Die Informationen hierfür werden aus *The Internet Movie Database* abgerufen. Wir untersuchen die wesentlichen Elemente, die für die Konstruktion eines Chatbots benötigt werden und analysieren verschiedene Ansätze, wie die Erinnerung des Gesprächskontexts gewährleistet werden kann und die auf regelbasierte Lösungen aufgebaut werden können.

Das Abrufen relevanter Informationen aus der relationalen Movie-Datenbank ist eine anspruchsvolle Aufgabe, da die natürliche Sprache auf eine SQL-Abfrage abgebildet werden muss. Dadurch wird die relationale Datenbank in ein RDF-Format umgewandelt. Dies ist eine graphenorientierte Datenbank, die semantische Sachverhalte enthält. Durch diese Umwandlung werden die Daten in einer Form repräsentiert, die der natürlichen Sprache ähnlicher ist. Dies macht das Abfragen einfacher.

Des Weiteren wird eine Lösung zur Erinnerung des Gesprächskontexts, einschliesslich des vorherig migrierten RDF-Formats vorgeschlagen. Die Metadaten der semantischen Datenbank werden benutzt, um Wörter eines neuen Satzes dem aktuellen Kontext zuzuordnen.

Die Kombination dieser zwei Lösungen erlaubt das generieren von qualitativ besseren Abfragen.

Aufgrund zeitlicher Einschränkungen und eines Mangels an geeigneten komparativen Datenbeständen, war es nicht möglich, die vorgeschlagene Lösung ausführlich zu evaluieren. Trotzdem war es möglich festzustellen, dass durch die Nutzung von Daten einer spezifischen Domäne in Form von Tripel, sowohl das Verständnis der Wörter als auch die Einsetzung der Ansätze zur Erinnerung des Gesprächskontexts vereinfacht wird. Ausserdem zeigte die Arbeit, dass die Auswertung von komplexen Verhältnissen in einem Graph rechenintensiv sein kann.

Um eine umsetzbare Lösung zu finden, ist eine Weiterentwicklung der Software notwendig.

Acknowledgements

We would like to thank our project advisor Dr. Mark Cieliebak for his support and availability, helping us throughout the whole semester. We also wish to thank Michael Longthorn for his linguistic feedback on the first draft of this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

DSL	**D**omain **S**pecific **L**anguage
ETL	**E**xtract **T**ransfer **L**oad
IRI	**I**nternationalized **R**esource **I**dentifier
NLU	**N**atural **L**anguage **U**nderstanding
POS	**P**art **O**f **S**peech
RDB	**R**elational **D**ata**b**ase
RDB2RDF	**R**elational **D**ata**b**ases to **RDF**
RDBMS	**R**elational **D**ata**b**ase **M**anagement **S**ystem
RDF	**R**esource **D**efinition **F**ramework
REfO	**R**egular **E**xpressions **f**or **O**bjects
SPARQL	**S**PARQL **P**rotocol **a**nd **R**DF **Q**uery **L**anguage
SPO	**S**ubject-**P**redicate-**O**bject
SQL	**S**tructured **Q**uery **L**anguage
SRL	**S**emantic **R**ole **L**abelling
UI	**U**ser **I**nterface

Glossary

Cardinality	The number of elements in a set or other grouping, as a property of that grouping
Entity	Identified subject, predicate or object of a triple
Intent	Objective of a question
Monkey-pacthing	Piece of code which dynamically extends other parts of code
Ontology	Set of concepts and categories in a subject area or domain that shows their properties and the relations between them[2]
Tagline	Enticing short phrase used by marketers and film studios to advertise a movie
(Web-) scraping	extract data from websites

[2] https://en.oxforddictionaries.com/definition/ontology

Chapter 1

Introduction

The thesis describes the endeavor to build a chatbot, as well as the design of the entire architecture.

Two open-source libraries for natural language understanding are utilized.
The first one is Quepy, a regular-expression-based solution to extract entities and intentions from a text. Its foundation relies on REfO, another open-source library from the same software company. REfO allows the writing of regular expressions using Python objects instead of the more typical strings.
The second library is Rasa NLU, an open-source alternative to more advanced proprietary services such as API.ai or Microsoft Luis. We opted for Rasa because we did not want to depend on closed-source solutions. A further advantage of this choice is being able to understand the background process from raw text to useful information, inclusive of elements such as intent and entities.

This leads to a considerable advantage over online services, as they do not allow direct integration within their system.

The source of information comes originally in plain text from The Internet Movie Database. In order to simplify the lookup of data, we imported it in a MySQL database. Since building queries concerning many joined tables represents a burdensome task, the relational database is migrated onto a RDF store.

Our work mainly focuses on the issue of context in a conversation. Often, a conversation in natural language contains references to previous interactions. Therefore, there is the need to recall the relevant information from the whole conversation and such information as became futile as the discussion progressed.

One soon becomes aware that understanding natural language understanding is a challenging topic. English, as with many other languages, displays many ambiguities, that is words which can assume various, different meanings depending on the sentence. Because of this ambiguity, it can be difficult to estimate their relevance in a specific language domain.

Valuable elements within sentences are identified by looking for matches in a knowledge graph, a semantic database which describes the relations of the data with short, minimalistic statements. These statements are called triples and are composed of subject, predicate and object. A knowledge graph is used to discover the relationships between various, unrecognized utterances.

Our proposed solution is limited to a specific domain and requires data to be stored as triples. The design of the relations between entities plays an important role, as we use this to relate entities to each other.

Our approach makes use of all the information available in a conversation. Thus, it does not limit or direct the flow of the discussion. On the other hand it restricts the range of topics. This can be solved by adding data taken from other domains to the knowledge graph.

Since our project is still in an experimental phase, we are not able to provide proven, scientific results nor comparisons with other tools. Nevertheless, some solutions are set out which can be refined in the future.

In the field of context-recalling, there are to date only a small number of workable methods.[1] Therefore, it would be reasonable and pragmatic to compare our results with other published works, since there is no standardization of test set.

1.1 Chatbot

The word comes from the union of the words chat, "a talk in a friendly and informal way"; and bot, "an autonomous program". The combination of the two signifies "a computer program designed to simulate conversation with human users".[2] The purpose of a chatbot can vary from conversations about general or specific topics, to the interaction with other devices; for example, by turning on the lights in a house or by playing some music through a stereo device.

In the last few years, many important companies, such as Google, Amazon, Apple and many others, started integrating chatbots in their products in order to improve the user experience.
Chatbots give the user a new form of interaction with the device or with the user-support by having a conversation in natural language. There is no need of having a human operating the chatbot; this allows a (theoretically) unlimited number of users to talk with the user-support.
Existing chatbots are able to answer questions such as "How long is the San Francisco bridge?" by retrieving information from Wikipedia or other sources.

1.2 Inception

Thanks to the rapid developing of new or more advanced technologies, people are able to communicate with the rest of the world in a completely different fashion in comparison to 50 years ago.
With the introduction of the Internet and smartphones, people can talk and chat with each other, overcoming physical boundaries.

The introduction of messaging apps, such as the first IRC clients, lead to the creation of machines able to understand the human language: bots. These softwares are designed to automate tasks which otherwise would require a person. In particular, chatbots are

[1] to the best of ability, we were not able to find a lot of scientific information
[2] https://en.oxforddictionaries.com/

integrated in chattings apps in order to handle user's requests via textual inputs. Audio inputs can be used as well but require a speech-to-text component.

The first chatbots were merely to handle single sentences. They looked for keywords and specific templates in order to evaluate an answer. "Yet, capability to trigger behavior and use additional functionalities in a context sensitive way constitutes one of the significant characteristics of commercial chatbots".[5, p. 8]

1.3 Point of departure

This chapter serves to provide a background understanding for the reader. The surroundings of the topic are explained, existing solutions are discussed and briefly evaluated.

1.3.1 Target audience

Intended audience is anyone who would like to dive in the world of chatbots and is interested in alternative approaches for building context-sensitive conversational agents. Additionally, ETL procedures to convert from relational datasets into a representation of resources through graphs is also a highly discussed topic in this thesis, and we believe it can provide an idea on the various DOs and DONTs when dealing with graph databases.

1.3.2 State Of The Art: existing solutions to the problem and their limitations

Whatismymovie

Whatismymovie[3] is not a chatbot, but seems to be able to understand natural language well. It has a simple interface, which is oriented to information retrieval and not to conversation handling. We consider this to be a great example for some clever UI-design decisions. Figure 1.1 represents the result of a search on the website.

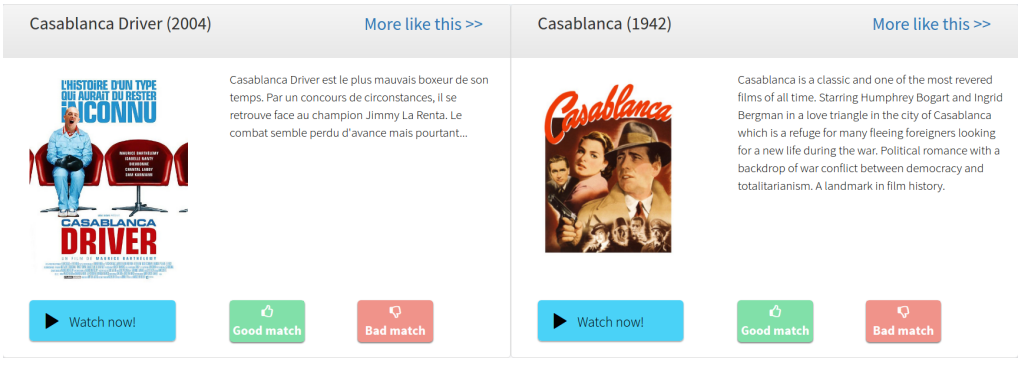


FIGURE 1.1: Snippet of the result of the research "Casablanca"

It has a standard interface, in which the results are displayed with pre-defined information, such as the title, release date, trailer, plot and other similar movies.

[3] https://www.whatismymovie.com/

One important feature is represented by a couple of like/dislike buttons, through which the user can provide feedback on the pertinence of the presented information. This feature is important, because it allows the developers to evaluate their product.
With a simple click, it is possible to "ask for more". We consider this to be smart UI-design, since the user does not have to explicitly ask for more results, but he is provided with the option to do so.

The research interface offers different types of matching, as presented in figure 1.2.

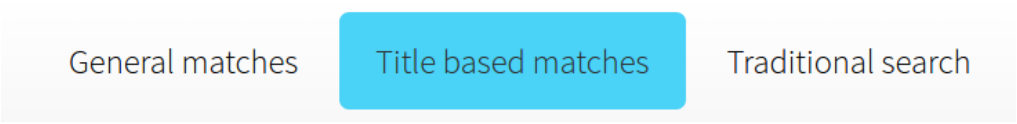


FIGURE 1.2: Snippet of the search options presented on the page

Since some functionalities are fixed, we assume it is due to the user's choice of clicking those items often. It is important to keep in mind that the server should also offer and propose on its own. This could activate a conversation,

Although not interactive, it is overall a good system from which a lot can be learned.

Google

Although *Google*[4] is also not a chatbot, it can do spectacular things, such as retrieving movies in which multiple people are starring, For example, figure 1.3 shows a snippet of the results retrieved automatically by Google, when requesting movies including Brad Pitt and Angelina Jolie.

FIGURE 1.3: Snippet of the search results presented on the page google.com

Similarly to *Whatismymovie*, the first things appearing are the title, the year of release and a cover image. These three pieces of information seem to be a standard among systems capable of answering movie-related questions. It is also important to notice how the information about the given filters (the names of both actors) are repeated in the output section. This is probably done to avoid confusion, in case actors with similar names get matched or a mistake in the retrieval process occurs.

[4]https://www.google.com/

1.3.3 Business perspective

Chatbots are becoming more and more important for the human being. In the last few years, people started working with them in a regular basis. Voice assistants, such as Apple Siri, Google Now or Microsoft Cortana are being used by an increasing number of users. Business Insider provides different studies which show how the use of messaging apps has increased in the last 15 years.[5] It resulted in a surpassing use of messaging apps over social networks.
Business Insider provides also a study regarding the desire of companies to have chatbots.[6] They predict that 80% of the companies will probably use chatbots to connect with their clients. The report projects potential annual savings in the order of billions of dollars.

1.4 Well-known problems

When building a chatbot, it is good practice to analyze well-known problems and look for existing working solutions instead of inventing new ones right away.

1.4.1 Input errors

The raw user's input may contain some grammatical or orthographic errors and could make the job of the chatbot more challenging. One possible solution is to parse the raw user's input into an auto-corrector in order to work with a cleaner text. Since our development has been a continuous experiment, we did not try to fix input errors but we expect the user to provide error-free texts.
A similar approach can be used when dealing with audio data as input. There are some words which have similar (or even equal) pronunciation but a totally different meaning. For example "bread" (/brɛd/) and "Brad" (/brad/), the first referring to the food and the second to the proper name of a person. A speech-to-text tool may not be able to deduce the right word in such situations. In this case it may be possible to build a tool which replaces a word from another topic (food) to one specific (actor) by looking at similarities of the pronunciation of multiple words.
We did not implement such a tool but we think it is interesting for those languages containing many words whose sounds are difficult to distinguish from each other.

1.4.2 Written vs. spoken dialogue

People may have noticed that written dialogues differ from spoken ones in different ways. Le Bigot et. al [6] have shown that the spoken language is usually more complex, as more words per-sentence are used. In addition, it is important to consider that the user may not understand what the chatbot says, and there could be the need of repeating the answer. This obviously does not concern written conversations.
For these reasons, we decide to focus on written dialogue only.

[5] http://uk.businessinsider.com/the-messaging-app-report-2015-11?r=US
[6] http://uk.businessinsider.com/80-of-businesses-want-chatbots-by-2020-2016-12?r=US

1.4.3 Ambiguity

We need to correctly understand a sentence in order to provide a valid answer. Ambiguity is not only a common problem in natural language processing, but also in discussion between humans. To better understand this issue, figure 1.4 provides some real-world examples from the book *Natural Language Processing with Python*.[7]

Word Sense Disambiguation

a. The lost children were found by the *searchers* (agentive)
b. The lost children were found by the *mountain* (locative)
c. The lost children were found by the *afternoon* (temporal)

Pronoun Resolution

a. The thieves stole the paintings. They were subsequently sold.
b. The thieves stole the paintings. They were subsequently caught.
c. The thieves stole the paintings. They were subsequently found.

FIGURE 1.4: Ambiguity examples

In the *Word Sense Disambiguation* case, the preposition *by* can assume three different meanings. In order to understand the sentence, we need to know the exact definition of the last word.

Pronoun Resolution is a common problem in languages where the subjects may not have a genre. In this case, *they* may refer to either *the thieves* or *the paintings*. In cases `a.` and `b.`, the verb provides the right information to understand the subject, as the thieves cannot be sold and the paintings cannot be caught. However, it is still required to understand the definition of the verb.
On the other hand, even knowing the meaning of the verb of case `c.`, it is impossible to comprehend who the subject of this sentence is. In this case, even humans cannot fully understand the second sentence. This problem is not present in languages where the pronoun *they* exists in different forms, and *the thieves* and *the paintings* are not of the same genre.
Even though we do not directly face this problem in our thesis, we expect the user to provide ambiguity free text.

1.4.4 Context ambiguity

Context ambiguity is present when it is not clear if a sentence depends on what has been said in the past. Figure 1.1 shows a context-handling problem we faced. The second sentence is sent after the answer to the first one has been received. The problem consists in understanding whether the user wants only the movies starring Tom Cruise *or* the movies of 2016 starring Tom Cruise.
One possible solution would be to ask the user for clarification, for example with: “Also the ones of 2016?”.

1.4.5 Orthographic and lexical mistakes

The raw input that the user provides with his keyboard may contains mistakes. If a system is expecting an error-free input, flaws are not accepted. One solution would be to

User says	Chatbot says
Show me the movies of 2016	
	`<MOVIES OF YEAR 2016>`
Only the ones starring Tom Cruise	
	`<??>`

Table 1.1: A context ambiguity example

pass the raw data through an auto-corrector. This would probably remove some mistakes made by the user but it may change some special words, such as personal names, to non-related terms.

1.5 Objectives

Our main goal is to build a bot prototype that is able not only to answer a series of questions, but also to interact with the user in order to make a conversation feel more friendly and human-like. In order to achieve such a result, we need to identify the crucial aspects of a dialog.
First of all, the bot has to deliver the right results regarding what it has been asked.
Second, the answer should be contained in an english sentence - or in the language of the user.
Finally, the bot should remember what has been talked about earlier, in order to understand what the current topic is. Furthermore, a good-looking app would enhance the user experience.

Some ideas of what a chatbot could actually do include:
finding similar actors to the user using image similarities techniques; checking the public opinion about a movie using text analysis of messages on Twitter; renting a movie directly on some movies-content providers (such as amazon and iTunes); book tickets to a cinema; rate a particular movie and receive film recommendations.

Chatbots are usually integrated within chatting apps such as Telegram[7], Facebook Messenger[8] or Slack[9].
There are multiple ways in which a user can interact with a chatbot. The first one is to have a human-like conversation with the bot; the user may ask "Show me the best movies of 2016" or "I feel like watching an action movie, please suggest me one".
The second option is to provide the user with possible answers; those can vary from simple yes or no to a list of available choices such as as a list of genres (eg. horror, thriller or drama). In the latter case, the user has less freedom of choice but this approach simplifies the task of the bot as no Natural Language Understanding (from here on referred as NLU) is required.
In order to improve the user-experience, the conversation should be bi-directional. This means that the user should not be the only one asking questions, but the chatbot could ask some on its own. We believe that a bi-directional conversation can enhance the user-experience as the bot becomes an active counterpart. Hence the bot has the possibility to:
request the user to reformulate the question in case it has not been understood correctly;

[7] https://telegram.org/
[8] https://www.messenger.com/
[9] https://slack.com/

ask for more specific details to improve the results of the answer; answer with a natural language text about the user's input.

1.6 Considerations

Given the complexity and extent of the subject of our project, combined with the limited time-frame available, we considered far-fetched to be able to implement a system which would produce astonishing results. These three factors influenced our design decisions remarkably and shifted our goals into favoring a malleable application, that could be utilized as a starting point for future projects, over a finished but static product. Instead of focusing on a single problem, we decided to invest our efforts into solving multiple small problematic aspects that compose a chatbot, with the objective of combining or linking them together in an efficient way. This approach decision lead to a modular system whose components can be replaced or extended without having to apply modifications on others.

Chapter 2

Technologies

This chapter illustrates the adopted technologies together with their purposes. The order of the sections does not represent the importance nor the relevance of the technology described.

2.1 REfO

Regular Expressions for Objects[1] is a Python library which permits writing regular expressions using objects instead of the more common strings. Table 2.1 show how to write regular expression using REfO:

Regular expression	REfO equivalent
`ab`	`Literal("a") + Literal("b")`
`a*`	`Star(Literal("a"))`

`(ab)+|(bb)*?`	`a = Literal("a")`
	`a = Literal("b")`
	`Plus(a + b) | Star(b + b, greedy=False)`

Table 2.1: Regular expressions in REfO

Listing 2.1 shows how named groups can be assigned to any sub-match and retrieved later on.

```
1 regex = Group(Plus(a + b), "foobar")  | (b + b)
2 m = match(regex, "abab")
3 print m.span("foobar") # prints (0, 4)
```

2.2 Quepy

Quepy[2] is a framework to transform natural language questions to database queries, developed by machinalis. The mechanism is based on regular expressions: the developer defines question templates that are then used and matched to an input to extract relevant pieces of information, such as intent and values.
Section 5.1 provides an in-depth look at Quepy, together with our use and improvements.

[1] https://www.machinalis.com/blog/refo/
[2] http://quepy.machinalis.com/

2.3 Rasa NLU

Rasa NLU is a new, open-source alternative to closed-source services such as API.ai from Google and Microsoft's LUIS. These services provide APIs to parse natural language to extract the intention and other important values.
Rasa NLU is a tool which exploits other free softwares for three tasks:

- Pre-processing
- Entity extraction
- Intent classification

The capabilities of this framework are probably more limited than the other two alternatives, but we believe that it can suit our needs better, thanks to its flexibility. Due to the architecture of the framework, it is possible to swap components that implement the three aforementioned features, an operation which is not possible with API.ai or LUIS.

2.4 SQL

Structured Query Language (or SQL) is a query language to interact and manage data contained in a relational database management system (RDBMS), such as MySQL. In this project, this language is mostly used in the process of transforming the IMDB from the relational to the RDF form.

2.5 SPARQL

SPARQL stands for the recursive acronym "SPARQL Protocol and RDF Query Language" is defined by the World Wide Web Consortium (W3C)[3]. It is a query language able to retrieve and store data that from and into RDF Triplestores, such as Apache Jena and GraphDB. At the time of this research, all our queries and the utilized triplestores make use of the version 1.1 of SPARQL[4].

2.6 R2RML

"RDB to RDF Mapping Language" is a language for expressing customized mappings from relational databases to RDF datasets[5]. We cover this mapping language more in depth in the chapter "Transition from relational to RDF database"

2.7 Triplestore

Triplestores are essentially graph databases storing facts in the form of triples. They describe the dataset by containing a set of triples, which are third degree tuples. For

[3]https://www.w3.org/
[4]https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/
[5]https://www.w3.org/TR/r2rml/

highly interconnected data, they represent a more flexible and less expensive alternative to relational databases. One of the key features of triplestores is inference: the ability of being able to uncover new information our of the existing relations[8].

Chapter 3

Data

In order to provide useful answers to the user, it is crucial to gather the right data for querying it easily. In this chapter, different movie's datasets available online are considered. Additionally, the transition from a relational database to a RDF is explained.

3.1 Sources

We primarily focused on retrieving details about movies but we also mention some dialogue datasets which can be used for modeling training.

3.1.1 Information about movies

There are many online sources, however, in order to simplify this task, one single dataset is used and keeping it up to date with additional information is not considered. Furthermore, to achieve more control, flexibility and lower response times, the data is stored locally. Thus, discarding online APIs to retrieve information about movies.

Wikipedia provides an enormous amount of up-to-date information but requires significant work to find and extract useful information about a specific topic. One first needs to know which pages of Wikipedia are relevant (eg. movies and actors), select only the useful content and store it in a consistent scheme.
In fact, DBpedia created its own ontology from Wikipedia using infoboxes, the top-right summaries found on many articles. Furthermore, DBpedia states that a issue consists of the "weaknesses in the Wikipedia infobox system, like having different infoboxes for the same class, using different property names for the same property, and not having clearly defined datatypes for property values"[9]. For this reason we discarded this choice.

Google and The Open Movie Database both offer online APIs. Google can already answer many movie-related questions. For example, when googling "Movies starring Tom Cruise", a list of related titles is shown, together with the poster and the release year. Apart from the fact that we cannot rely on these services only, OMDB[1] is also not freely available. In addition, OMDB used to provide a monthly dump but because of DMCA Takedown Notice this functionality is not offered anymore since 10.12.2016[10].

[1] https://www.omdbapi.com/

Youtube is a valid resources for providing trailer of movies. It also features a set of useful APIs. We can then embed the resulting URL into an iFrame (as shown in section 4.2.2) in order to render the video directly in the chat. This solution is much more practical than storing the videos offline as the files are much larger than the textual data, which would then require additional bandwidth.

3.1.2 The Internet Movie Database

IMDb provides a subset of the data in form of text files[2]. This collection contains a lot of information about movies, people and companies and is split in about 50 files. Table 3.1 shows the names and sizes of the first 8 compressed text files.

Filename	Size
actors.list.gz	329.2 MB
actresses.list.gz	185.4 MB
aka-names.list.gz	8.9 MB
aka-titles.list.gz	10.1 MB
alternate-versions.list.gz	2.7 MB
biographies.list.gz	214.1 MB
business.list.gz	11.7 MB
certificates.list.gz	6.2 MB

Table 3.1: First 8 IMDB compressed files

The size of the complete text dataset sum up to 1.95 GB compressed. Since retrieving specific data from text files is not trivial, we imported the dataset into a MySQL Database. IMDBpy[3] is a tool which offers parsing the IMDB text files into a SQL Database. The procedure is straightforward but the actual import needs quite a lot of time. It required 322 minutes, where 40 were necessary for writing the data, 264 for creating the foreign keys and 17 for the indexes. This operation was executed on a laptop with an Intel Quad-Core i7-4710MQ processor, 8GB RAM and a SSD.

This tool creates the entity-relational schema shown in figure 3.1. Some tables require clarification:

- `title` contains the movies and shows
- `name` contains the people
- `info_type` contains all sort of information's types, such as: *budget*, *genres*, *plot* and *votes*[4]
- `movie_info` and `movie_info_idx` contain the actual information, either as a string in the first case or a number respectively

The SQL dump of this database has a size of 7.38 GB (1.95 GB compressed in .gz).

Note: This dataset can ONLY be used for personal use[11].

[2] http://www.imdb.com/interfaces
[3] http://imdbpy.sourceforge.net/index.html
[4] There are 113 types of information available

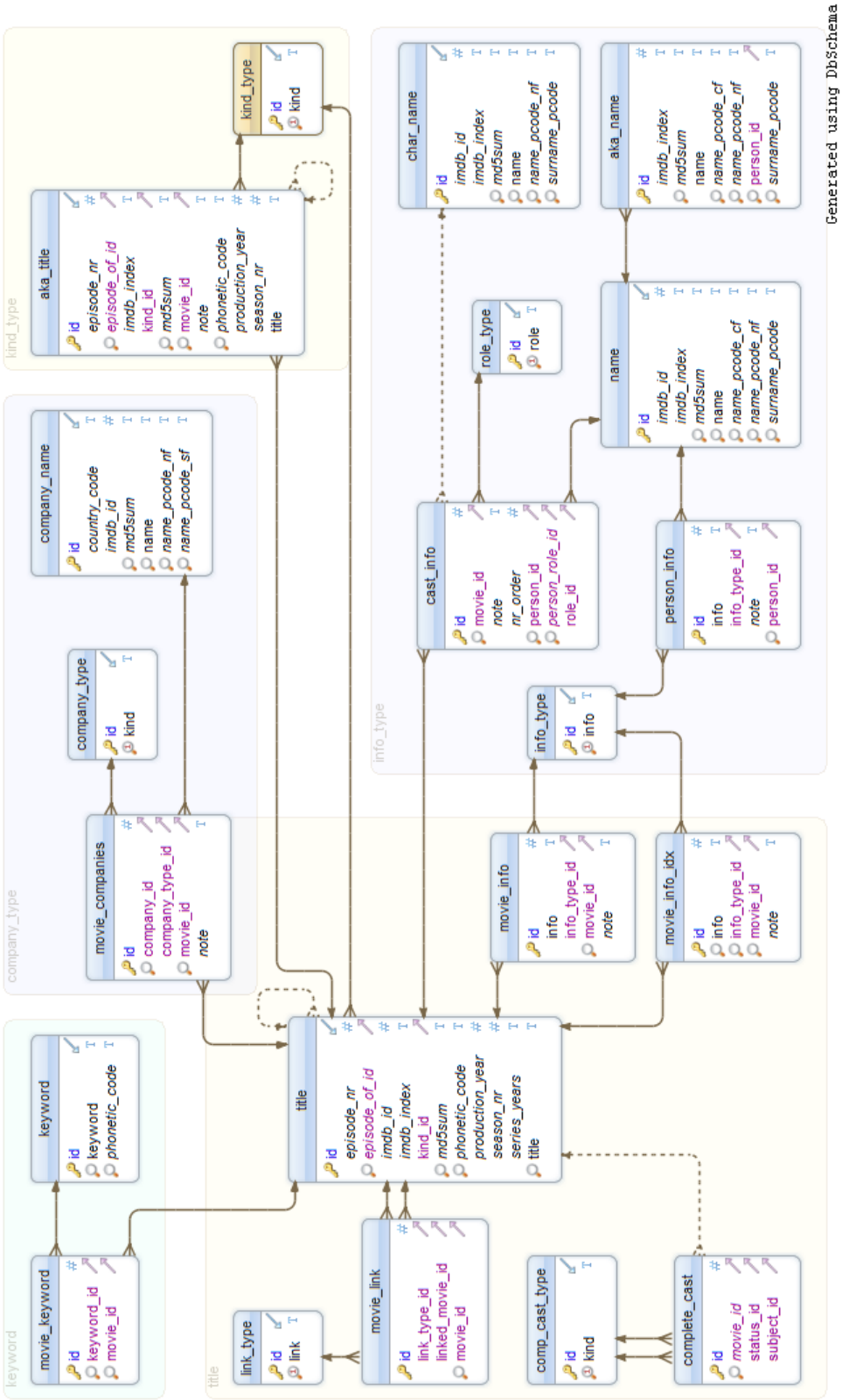


FIGURE 3.1: Relational schema of The Internet Movie Database[1]

3.1.3 Dialogues dataset

Facebook bAbI project includes multiple datasets concerning dialogues[5]. It contains specific datasets about movies for evaluating the three tasks: question-answering, recommendations and question-answering with recommendations. For instance, the three sets look like this:

User says	Chatbot says
What does Grégoire Colin appear in?	
	Before the Rain
Movies X, Y, Z are movies I really liked. Can you recommend a movie?.	
	Mulholland Drive
I liked X, Y, Z. I am looking for a drama movie.	
	Dogville
Who is that directed by?	
	Lars von Trier

Table 3.2: Facebook bAbI data samples

While this dataset may seem useful for data retrieving, it does not contain relations between anything. For instance, it is not written anywhere that the director of Dogville is in fact Lars von Trier. Thus, we believe it cannot be directly used for solving our problem. In addition, the range of questions is very limited. We think that this dataset could be in part used but it requires some work since the examples are not classified (i.e. the intent is not defined).

3.2 Relational database to RDF

In this section the important aspects concerning the transition from the relational database of IMDB to a triplestore solution are explained. The motivations behind this choice together with its advantages and disadvantages are covered. Furthermore, we illustrate the various steps involved to achieve a successful transition. We provide the solutions to some problems which appeared during the transition, such as RAM limitation and slowness regarding the import of the data into the triplestore.

3.2.1 What is a RDF store?

A RDF store is a database for storing and retrieving triples. Before explaining what a RDF store is, we would like to do a small recap on some key aspects of relational databases. A relational-database, like MySQL, stores the data in different tables, each of them having multiple columns and rows (the latter are identified by a unique key). The tables are referenced to each other with so called relationships, where the unique key of an element is referenced from another linked table. Figure 3.2 shows an example of a relational database containing two tables.

[5]https://research.fb.com/downloads/babi/

Person

id	name	familyName	birthDate	nationality
p10	Daniel	Craig	April 13, 1968	n1
p11	Angelina	Jolie	June 4, 1975	n3

Nation

id	nation
n1	Great Britain
n2	Switzerland
n3	United States of America

FIGURE 3.2: Example of a two table relation

Every person (*row*) in the table *Person* has a reference to another row in the table *Nation*. The reference is done by storing the id of the nation in the person.

A RDF graph on the other hand, is a set of 3-elements tuples, called triples. A single triple consists of three components (shown in figure 3.3):

- **Subject**: an IRI[6]
- **Predicate**: usually a compacted IRI (more below)
- **Object**: an IRI, a literal or a blank node

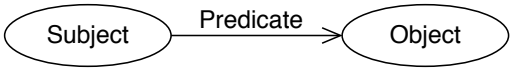


FIGURE 3.3: Subject Predicate and Object in a RDF Graph[7]

Figure 3.1 shows a simplified raw triple file from our RDF graph.

```
1 @prefix dbp-owl: <http://dbpedia.org/ontology/> .
2 @prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
3 @prefix dbpprop: <http://dbpedia.org/property/> .
4
5 <http://chatbot.com/movie/1122> rdf:type dbp-owl:Film ;
6 <http://chatbot.com/movie/1122> dbpprop:name "Gran Torino" ;
7 <http://chatbot.com/movie/1122> dbpprop:starring <http:///chatbot.com/
    person/3344> ;
```

LISTING 3.1: Example of a raw triple file

The keyword `@prefix` defines a prefix name, which refers to a part of IRI and it is used for compactness and readability. For example, the property `dbpprop:name` stands for `http://dbpedia.org/property/name`. In this example we have defined 3 prefixes (`dbp-owl`, `rdf` and `dbpprop`) and 3 triples sharing the same subject, denoted by the long IRI "http://data.example.com/movie/1122". The first triple defines that the subject with id `1122` is a film, by making use of the `rdf` prefix and the property name `type`. The second one defines the title of the previously mentioned film, which is the literal object "Gran Torino". The last triple represents the connection between the film (in this case "Gran Torino") and the person with id `3344`, of which if we had to inspect its name and family name properties, would turn out to be *Clint Eastwood*, who also starred in the movie. The triples permit a more natural way to define related data. Table 3.3 shows the same examples as above, together with other cases.

[6]The Internationalized Resource Identifier (IRI) is used to identify resources, in a very similar way to the more common URI, but it allows the use of non-ASCII character as well https://www.w3.org/International/articles/idn-and-iri/

[7]Image taken from: https://www.w3.org/TR/rdf11-concepts/

Subject nr.	Predicate	Object
1122	*is of type*	Film
1122	*has the title*	Gran Torino
1122	*has been rated*	8.2/10
1122	*has the actor*	subject nr. 3344
3344	*is of type*	Person
3344	*has the name*	Clint
3344	*has the family name*	Eastwood
3344	*was born in*	May 31, 1930

TABLE 3.3: Examples of triples in natural language

An additional representation is shown as a graph in figure 3.4.

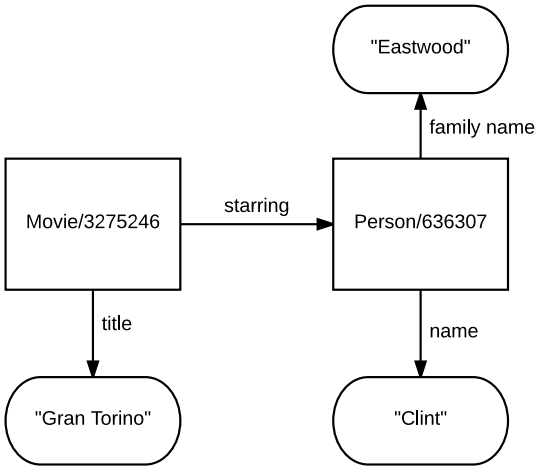


FIGURE 3.4: Example of triples inside a RDF graph

3.2.2 Relations and terminology

In order to be consistent with the DBpedia ontology[8] we adopted the same terminology of relations in our dataset. For example, an actor has the relation *dpprop:name* and *dpprop:familyName*[9]. This was done with the purpose of being able to adapt to existing datasets such as the one provided by dbpedia[10]. However, some of their design decisions were sub-optimal and would have lead to dissociate of certain information, such as the release date and location of a movie.
A movie can be released on multiple combinations of date and location,for example, due to the time needed in order for the translators to dub it. Therefore to allow a correct modeling, both the date and location of the release need to be properties of a single separate class: release. However, the designers of the dbpedia ontology did not consider this case initially and decided to insert a note on the description of both single, flawed properties: "releaseLocation: Usually used with releaseDate, particularly for Films. Often there can be several pairs so our modeling is not precise here..." [12].

[8] http://mappings.dbpedia.org/server/ontology/classes/
[9] *dbpprop* stands for DBPediaProperty
[10] http://wiki.dbpedia.org/Datasets

Example of complex relations

Actors can participate in multiple movies and movies can have many actors as well. This is a case of a many-to-many relationship. Figure 3.5 shows how this relationship would be handled in a typical relational database approach, which is, by creating a junction table.

Actor ID	Name	Married to
1	Angelina	2
2	Brad	1

(A) Table Actor

Movie ID	Name
70	Fight Club
33	Tomb Rider

(B) Table Movie

Actor ID	Movie ID
1	70
2	33

(C) Junction table Actor-Movie

FIGURE 3.5: Relational schema about actors and movies

Although it is a simple example, we can already notice that an overall understanding of the whole database is necessary in order to well comprehend the connections between values and tables.
As shown in figure 3.6 the same data is much more readable and intuitive under this structure. An additional advantage with this representation is that movie 33 and movie 70 have no direct connections with each other. They are however linked by the two married starring actors the two movies.

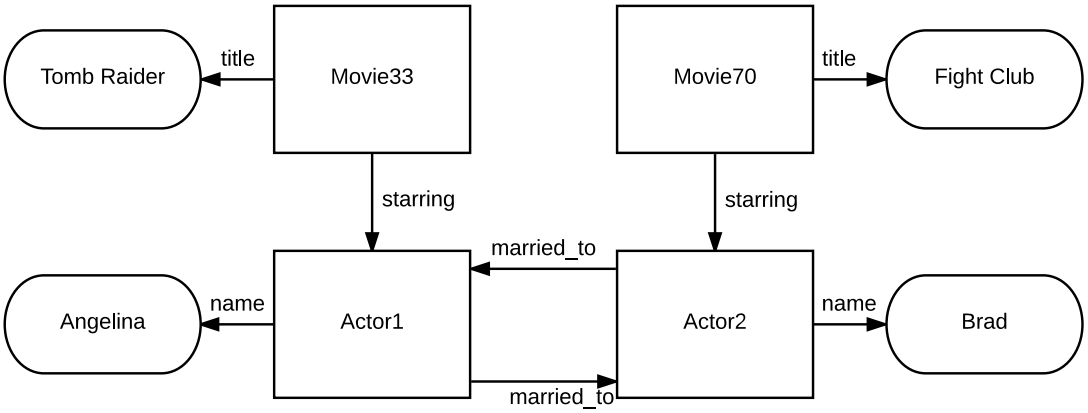


FIGURE 3.6: Example of triples inside a RDF graph

(See appendix B for the RDF counterpart of IMDb.)

3.2.3 Motivations

There are several reasons why we think that an approach using RDF databases would be a more viable option in our research. The motivations emerged during the early phases of the research when we started utilising quepy. (For a more detailed explanation of Quepy, check section 5.1).

D2RQ: A lost cause

D2RQ[11] is a solution to convert SPARQL queries to SQL on-the-fly. Using the Apache Jena API, D2RQ is able to access the underlying relational database through a virtual, read-only and transient RDF graph.

Limitations D2RQ only works with Jena. This fact, combined with having to forcefully operate on a static, unmodifiable version of a graph was not appealing.

Although this system has appeared to perform faster than normal SPARQL queries, the credibility of the tests has been questioned in the past[13].
Moreover, D2RQ only allows for a direct mapping and not for a custom one, i.e. it defines how columns and keys should be mapped to objects and predicates respectively. Thus, the RDF schema is restricted to being a copy of the schema structure of the relational database. Consequently, the RDF schema might be influenced from the (potentially bad) design decisions of previous relational database modelers.

For there reasons, we decided not to opt for D2RQ.

Not misusing Quepy

Quepy builds SPARQL queries from structured information which represents the relations between subjects and objects. Thus, it is not easily possible to transform this data into SQL queries, as they do not follow a similar grammar and syntax to the one used in SPARQL queries. Quepy is built with the concept of representing information with the structure of a triple, i.e. it generates SPARQL-queries with the help of predefined metadata. The relations between entities and literals is on the relations between types of entities and literals. Therefore, it is certainly more intuitive to utilise this framework the way it is meant to be used: for RDF graphs. Furthermore, in order to build SQL-queries from the data structure that quepy generated, monkey-patches on different parts of the library were needed and the resulting code was very unpleasant. This was accompanied by the realisation that, in general, writing SQL-queries from unstructured information was challenging, therefore we decided to try another approach.

Triple statements are similar to natural language

Although a relational representation might be less expensive regarding lookup time, a triple design is intuitively more similar to the structure of a sentence, where words represent paths and connections between the various subjects, predicates and objects.
For instance, figure 3.7 shows how the question “Show me comedy movie titles starring Will Smith” is mapped into a graph. The left part of the graph represents one part of the question, in this case “comedy movie titles”. The right part represents the subject “Will Smith” and the verb “starring” indicates how the left and right part are connected. Of course, identifying to which elements of the graph each word matches is not trivial and is indeed in the scope of the project. The value corresponding to the title of the movie, linked by the predicate *dbpprop:name*, is set to x0 (unknown value) because it is the target of the question.

[11] http://d2rq.org/

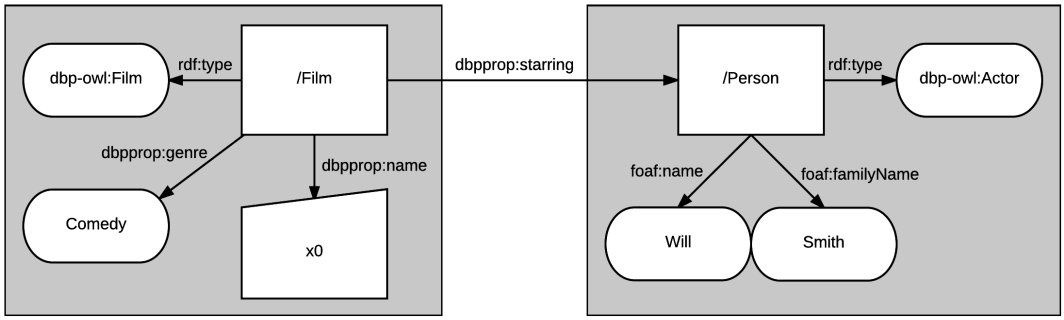


FIGURE 3.7: Representation of the words as paths through elements of the graph

NULL values are not required in RDF graphs

An additional reason for moving to RDF datasets is the dealing with NULL-values, a notorious issue in the world of relational databases. All the records need to contain a value in each column, even though they do not have the information at all. In this case, they assume the value NULL.
In RDF graphs, however, subjects not possessing any value for certain attributes simply do not contain these attributes at all. Therefore it is much easier to operate with non-existing values on RDF graphs, because they simply do not exist instead of taking any "magic" value (such as NULL or 0). For the same reason, it is also easier to add new data to the dataset, as subjects are not part of tables and therefore are not restricted to any schema design decision.

Because of the reasons mentioned above, we concluded that the conversion to a RDF store would solve part of the problems in the generation of queries. It will definitely reduce the complexity of the query because of its simplicity compared to relational databases.

3.2.4 Approach for the conversion

At the ISWC (International Semantic Web Conference) that took place in 2013, a tutorial presented an overview of the newly-defined W3C RDB2RDF standards: Direct Mapping and R2RML[2]. The session included a series of case studies from open-source contributors, in which people were able to successfully transfer huge amounts of data (approximately 150 million triples) from their relational databases into raw triple files.
In this section, we discuss the transformation from the relational dataset to a triple representation. The raw triples are represented with the Turtle format, based on the R2RML mapping standard.

The R2RML standard

The R2RML standard is described as "a language for expressing customized mappings from relational databases to RDF datasets"[2], and is one of the two languages of the RDB2RDF collection recommendation[14]. The word *customized* implies the possibility of writing mappings which do not simply represent the structure (schema) of the relational database mirrored onto triples, but can also adapt to any either newly defined or already

existing ontology. This is especially powerful because it allows to distantiate from design decisions that were driven by performance and integrity reasons in the world of relational databases, such as the well-known third normal form normalization. The importance of this factor is stressed in the chapter 3.2.3.

The conversion procedure creates a RDF graph out of the following input:[2]

- Relational database (schema and data)
- Target ontologies, a set of terms used to describe a domain
- R2RML Mappings between the relational database and target ontologies

Turtle stands for *Terse RDF Triple Language* and is a textual syntax for writing RDF graphs "in a compact and natural text form, with abbreviations for common usage patterns and datatypes. Turtle provides levels of compatibility with the N-Triples format as well as the triple pattern syntax of the SPARQL W3C Recommendation"[15]. The file extension for Turtle files is *.ttl*[12]. As their creators state, the syntax is intuitive to comprehend and will not be explained any further in this thesis.

Mappings in R2RML are written in Turtle format. A mapping file contains at least one (Triples-)Map, which itself contains exactly one definition of a logical table, one subject map and an arbitrary amount of predicate-object maps. Figure 3.8 shows in a graphical way the format of the file.

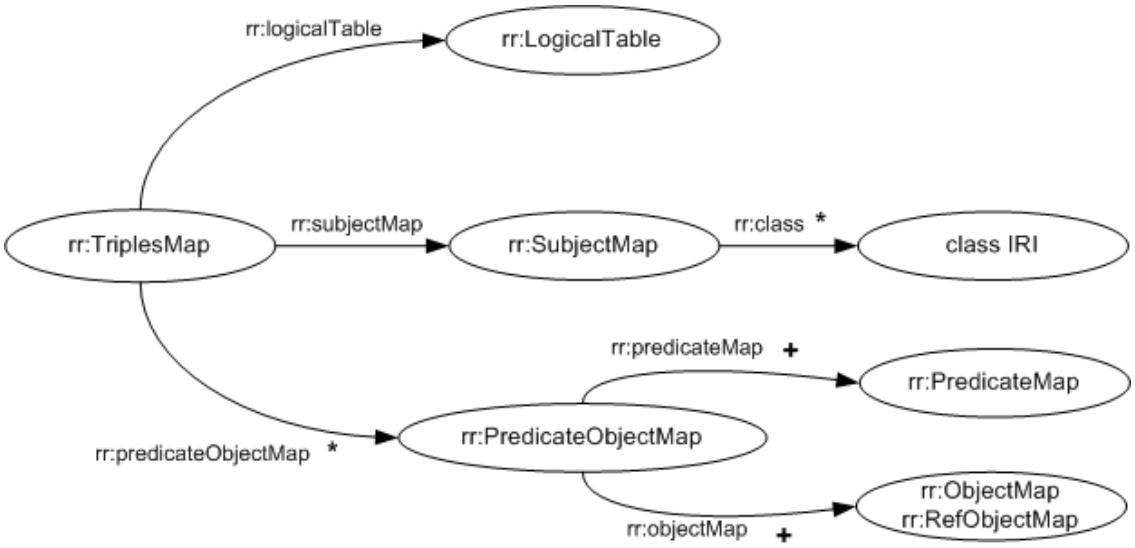


FIGURE 3.8: Properties of triples map
[16]

The plus symbol (+) signifies that the element must occur at least once, while the asterisk (*) signifies that it may occur once, multiple times or none at all.

[12]In this thesis, Turtle is also often referred as TTL.

Example: Assume one would like to convert the relational table shown in table 3.4, into the RDF graph shown in figure 3.9.

Id	Name	Gender
1	Alice	F
2	Bob	M

Table 3.4: Example of a relational table Person [2]

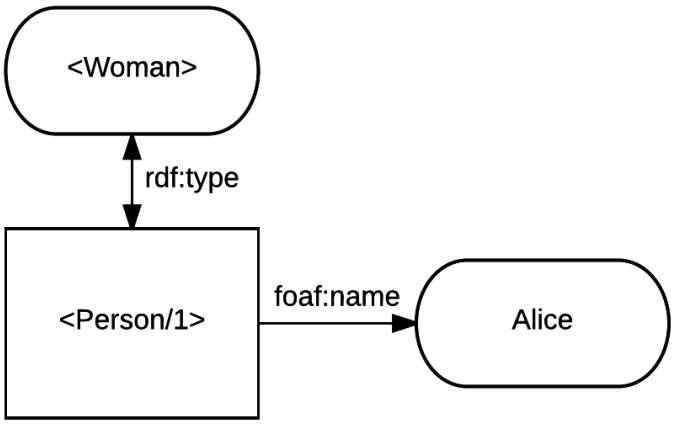


Figure 3.9: RDF graph converted from the relational table Person [2]

Listing 3.2 shows how the mapping is defined.
`rr:logicalTable`, together with `rr:sqlQuery`, define the SQL query that will extract the necessary information from the relational table *Person*. In this case, only women are retrieved, thanks to the `gender = "F"` filter.
The construct `rr:subjectMap` defines the IRI of the resource with `rr:template`, while `rr:class` will be converted into a `rdf:type` predicate, which defines the type of the subject as a woman.
Interesting to note is the syntax `{ID}`, that refers to the value of the selected relational column `ID`, and thus create a unique IRI.
Finally, `rr:predicateObjectMap` defines a predicate `foaf:name` pointing at the value stored in the selected column `NAME`.

```
1  @prefix rr: <http://www.w3.org/ns/r2rml#> .
2  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
3
4  <TriplesMap1>
5    a rr:TriplesMap;
6
7    rr:logicalTable [ rr:sqlQuery
8      """
9      SELECT ID, NAME
10     FROM Person
11     WHERE gender = "F"
12     """];
13
14   rr:subjectMap [
15     rr:template "http://www.ex.com/Person/{ID}";
16     rr:class <http://www.ex.com/Woman>
```

```
17    ];
18
19    rr:predicateObjectMap [
20      rr:predicate foaf:name;
21      rr:objectMap [rr:column "NAME" ]
22    ].
```

Listing 3.2: Example of a simple mapping [2]

Note: There are too many syntax elements that compose the whole standard, but only the most relevant to the approach discussed in this thesis will be explained. For a more thorough explanation, please refer to the official documentation.

R2RML Parser

is an open-source tool[13] for mapping a relational database into triples and it is based on the R2RML Standard[14] from the W3 Consortium.

Important files There are three files which are essential for the functioning of the tool: *r2rml-parser.sh*, *property files* and *mapping files*.

r2rml-parser.sh is the entry point of the tool and requires two parameters:

- `-p`, signifies that the next parameter will be a property file
- `<filename>.properties`, where `<filename>` is the name of the property file which contains information regarding the mapping file

The content of the bash script can be explored in listing 3.3.

```
1  echo "This is R2RML Parser 0.8-alpha. Run with -h for help on options."
2  echo ""
3  echo "Dumping from property file \$2 ..."
4  java -Xms1024m -Xmx1024m -cp "./*;./lib/*;" -jar lib/r2rml-parser-0.8.jar
       \$1 \$2
5  echo ""
6  echo "R2RML Parser 0.8-alpha. Done dumping \$2."
```

Listing 3.3: Content of r2rml-parser.sh

The parameters `-Xms` and `-Xmx` specify the minimum respectively the maximum amount of memory that will be allocated by the JVM upon start of the `.jar` executable.

Property files are denoted by the extension *.properties* and are used to the parameters which determine the modus operandi of the parsing tool. They define which mapping file must be loaded, in which format it is written, the output format and destination, the source database on which the queries will be executed and some other information of secondary importance.

The property file used in production for the creation of the actors in the rdf graph is shown in listing 3.4.

[13] https://github.com/nkons/r2rml-parser v0.8
[14] https://www.w3.org/TR/r2rml/

```
# Mapping source file and format
mapping.file=mappings/actor.ttl
mapping.file.type=TTL

# Output destination and format
jena.destinationFileName=outputs/actor.ttl
jena.destinationFileSyntax=TTL

# Relational DB source
db.url=jdbc:mysql://127.0.0.1:3306/imdb
db.login=<user>
db.password=<hidden>
db.driver=com.mysql.jdbc.Driver

# You can leave these
default.forceURI=true
default.namespace=http://chatbot-zhaw.com/base#
default.verbose=false
default.log=status.rdf
default.incremental=false

# Jena settings
jena.storeOutputModelUsingTdb=false
jena.tdb.directory=outputs

jena.showXmlDeclaration=false
jena.encodeURLs=false
```

Listing 3.4: Content of actor.properties

3.2.5 Problems with the parser

arose from the very beginning. These issues were discovered during the proceedings and influenced the approach taken to convert the relational database into a RDF graph. Such issues vary from lack of RAM space to some standard features (although very particular) not having been implemented.
In the following paragraphs the various approach ideas together with the problems that were encountered at that stage and their respective solution.

RAM Problem

The initial thought was to have a single mapping file, containing a SQL query with joins on all tables necessary to model the entire RDF graph. However, from the very first attempts on our laptops, the tool yielded an error message regarding the Java GC (Garbage Collector) being overflowed, as displayed in listing 3.5. (We are working on a machine with 8GB RAM.)

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit
    exceeded.
```

Listing 3.5: R2RML - Out of memory exception

Attempt: Increasing maximum allocable RAM

A brief look at the bash script revealed that it was possible to increment the maximum amount of allocable RAM memory, as shown in listing 3.3.

The maximum memory allocation pool was then increased to 6.5 GB for tests, by changing the content of the bash script at line number 4, where the call to the .jar file is made. Listing 3.6 shows this in practice.

```
1 java -Xms1024m -Xmx6500m -cp "./*;./lib/*;" -jar lib/r2rml-parser-0.8.jar
    \$1 \$2
```

Listing 3.6: R2RML - using 6.5 GB of RAM

After this modification, the bash script would be running for a few minutes longer in comparison to the previous run, then it would stop with the same exception error.

Consequently, we opened an issue on the github repository of the tool, where a discussion with the owners of the program started.[15] On the thread, a few suggestions were made:

- use a machine with more RAM
- divide the dataset in many, smaller chunks

The developer said that the current version of the tool keeps everything in memory, before storing the results on the disk.

Attempt: DB2Triples as alternative to R2RML Parser DB2Triples is another Java tool for parsing a relational database into triples. Instead of storing everything in memory, DB2Triples gradually saves the results on the disk. Even though the approach could be practical and scalable, the efficiency of the tool is very low. In three minutes, the tool was able to store only 90 triple statements. Since our dataset will contain million of triples, this tool has been discarded.

Attempt: Using a cluster with more RAM We received access to a cluster, in which we could reserve up to 128GB of RAM. Even though the available RAM would have been enough to store the complete dataset, the tool was not able to perform the transition. We believe that this is caused by a poor programming design of the tool.

Problems with the mappings The user nkos on Github suggested to split the mapping in multiple files. In our first attempt, we had one single mapping file per resource type: one to extract all information about movies, another one for the actors, and so on. Due to the expense of the join operations in the SQL queries, and the amount of data selected by these queries, splitting the mappings was thought to give better results.
For this reason, the mappings have been split in many more files which can then be used singularly, reducing the RAM necessary. Nevertheless, the RAM consumption was still too high four our laptops.

[15] https://github.com/nkons/r2rml-parser/issues/24

Splitting mappings and property files Every mapping template depends on a very specific property file. Creating these property files can be tedious, and thus we wrote a simple script which can automatically generate them. Listing 3.7 shows how the script works.
When the script is executed, it look at all the mappings file inside the `mapping/` directory and creates a `.properties` file for each of them.

```
#!/bin/bash

echo "This script generates properties from mappings files in the mappings / folder"
echo "Example usage: ./generate_properties.sh sql_db_name"
echo "Generating property file for mapping"

db_name=\$1

# use default name if param was not passed
if [ -z "\$1" ]; then
  db_name="imdb"
  echo "WARNING: No db name supplied. Using default 'imdb'"
fi

i=\$(date)
# For every file in mappings/ generate a properties file in properties/ with the same name
for mappath in mappings/*.ttl; do
  mapfile=\$(basename "\$mappath")
  mapfile_name=\${mapfile%%.ttl} # removes extension
  echo "#Mappings
        mapping.file=mappings/\$mapfile
        mapping.file.type=TTL

        # Destinations
        jena.destinationFileName=outputs/\$mapfile
        jena.destinationFileSyntax=TTL

        # Mysql
        db.url=jdbc:mysql://127.0.0.1:3306/\$db_name
        db.login=<user>
        db.password=<password>
        db.driver=com.mysql.jdbc.Driver

        # You can leave these
        default.namespace=http://chatbot-zhaw.com/base#
        default.verbose=false
        default.log=status.rdf
        default.forceURI=true
        default.incremental=false

        # Jena settings
        jena.storeOutputModelUsingTdb=false
        jena.tdb.directory=outputs
        jena.showXmlDeclaration=false
        jena.encodeURLs=false"
          > properties/\$mapfile_name.properties
    done
echo "Generated files for \$i mappings"
```

Listing 3.7: Content of generate_properties.sh

Using R2RML with multiple mappings Once the properties file have been generated, it is possible to execute R2RML parser. As mention earlier in this chapter, the script `r2rml-parser.sh` can handle one single property file at the time. Since we have more than 30 property files, we wrote a simple script `dump-all.sh` which executes R2RML Parser for each property file in the `properties/` folder.

Dynamic predicate Writing mappings is a tedious task and sometimes it is not very practical. We soon realized that some mappings look very similar to others. For example, the class movie has many different predicates referred to people, such as actors, directors and so on. Because of how the relational database is designed, it is simple to create a query which returns movieId, relation and person.
With the technique of having split mapping files, we also thought that it was a clever idea to map all the roles that someone might have in a movie just once. In fact, the idea is clever, since the subject type could be inflicted dynamically (by using the PredicateObjectMap with the predicate `rdf:type` manually, instead of using `rr:class` in the SubjectMap). The only difference between the mapping files of various person roles would have been in fact, just the predicate connecting the film to the person with that specific role. For example, movie points at actor with `dbpprop:starring`, whilst to point at the director it has to use `dbpprop:director`. So we thought that we could use `rr:template["dbpprop:{role_type_predicate}"]` to dynamically assign the predicate based on the role of the cast. Exactly like we did for the subject! Isn't it a great idea? However, R2RML Parser has a bug in which the resulting end file is empty. Therefore we had to create a mapping file for every cast role connected to the movie: movie_producer.ttl, movie_starring.ttl, movie_director.ttl, movie_setDesigner.ttl, ... This was not very pretty, but was the only option.

Property file option forceURI and mapping clause rr:IRI The option forceURI=true in the property file forces all URIs found in the `rr:objectMap` clause of a mapping file to be mapped as IRIs. Without this option, all IRIs mapped in the `rr:objectMap` clause through a `rr:template` statement would be considered strings. Consequently, all the connections between resources would be futile, as the RDF graph would consider the objects as being strings literals and not resource identifiers (i.e. IRI). A cleaner (and safer) alternative is the use of the clause `rr:termType rr:IRI;` in the `rr:objectMap` clause, which specifies that the resulting URI should be mapped as such, thus identified as an IRI by the RDF graph.

Listing 3.8 shows a mapping file with no `rr:termType rr:IRI` statement.

```
1  rr:subjectMap [
2      rr:template "http://data.example.com/movie/{MOVID}";
3  ];
4
5  rr:predicateObjectMap [
6    rr:predicate dbpprop:starring;
7          rr:objectMap [
8                  rr:template "http://data.example.com/person/{PERID}";
9          ]
10 ].
```

LISTING 3.8: Example mapping with *forceURI = false* and no *rr:termType rr:IRI* statement

```
<http://data.example.com/movie/2966092> dbpprop:starring
  "http://data.example.com/person/150591" .
```

Listing 3.9: Example of the resulting TTL file with *forceURI = false* and no *rr:termType rr:IRI* statement

To note in listing 3.9 the quotes around `http://data.example.com/person/150591`, which cause the object of the triple to be interpreted as a string and not as an IRI identifying a resource. Important aspect to be aware of is that the subject URI gets mapped automatically as IRI, without needing to specify the `rr:termType rr:IRI`. Justification for this lies in the syntax of RDF graphs: the only value that subjects can assume in RDF graphs are IRIs.

Setting `forceURI=true`, inserting the statement `rr:termType rr:IRI;` or doing both actions will cause the same result. That is, the object URI will be mapped and considered as resource IRI. This is the result wanted, since it permits to connect resources. Listing 3.10 shows the same example mapping as listing 3.8, only including the `rr:termType rr:IRI`.

```
rr:subjectMap [
    rr:template "http://data.example.com/movie/{MOVID}";
];

rr:predicateObjectMap [
        rr:predicate dbpprop:starring;
        rr:objectMap [
            rr:termType rr:IRI;
                rr:template "http://data.example.com/person/{PERID}";
        ]
].
```

Listing 3.10: Example of mapping file *forceURI* either *true* or *false* and *rr:termType rr:IRI;* present

```
<http://data.example.com/movie/2966092> dbpprop:starring
  <http://data.example.com/person/150591> .
```

Listing 3.11: Result of mapping with set to *forceURI=true* or *rr:termType rr:IRI* present

In listing 3.11 it is possible to note now for the object of the triple the typical `<` (lower) and `>` (greater) symbols, which stand to signify that the enclosed URI is in fact an IRI, a resource identifier.

rr:IRI is safer than using the option `forceURI=true` because the first just enforces one single object to be mapped as IRI, while the latter does the same for all the objects which contain URIs in the mapping file. In case an object of a resource would represent an URI which is not the IRI of a resource, such as the URL of a website, for example, it would faultily get mapped as an IRI.

Errors in the SQL queries are not noticed by the tool Sometimes R2RML Parser does not provide any information when something goes wrong. We faced many issues that were not reported and we invested a lot of time looking for them.
For example, if the column name in a mapping did not match any column of the SQL result set, no message errors were shown and the tool kept running, like if there were no issues.

Problem with the data itself When we loaded the triples into Jena, we soon realized that the dataset contains adult movies and other inappropriate keywords. For some reasons, such movies cannot be easily found on IMDb and were unnoticed until this phase. To solve this problem, we added some filters in the SQL queries used for the mappings. All movies with genre "Adult", "Erotica" and "Sex" have been excluded. Even after we filtered these genres, we noticed that some inappropriate keywords were still contained. We agreed that some keywords such as "sex" could be legit tags for a non-pornographic movie, but others were definitely inappropriate. The reason for this is that some pornographic movies were labeled with the genre "short" instead of the others mentioned above. To simplify this task, we decided to extract only movies, and exclude all tv series, documentaries and shorts.

Another issue that we faced concerns the number of actors per film. The dataset contains (almost) all the actors of every movie, even the ones which are not popular at all. This caused a difficulty in extracting interesting data from the dataset. Since our RDF schema did not contain the importance of an actor for a specific movie, we decided to keep only the ones with a importance lower than 10 (1 is the most important).

3.2.6 Triplestores: storing and querying the dataset

Two different database have been tried to store the triples. The first one is an open-source solution called Apache Jena Fuseki[16], while the second one is Ontotext's GraphDB[17].

Apache Jena Fuseki

Apache Jena Fuseki, from now on referred simply as *Jena* or *Fuseki*, is a SPARQL 1.1 end-point accessible over HTTP. It provides REST-style interaction with the RDF data and a web-accessible interface, from which the datasets can be loaded and and queried. For this project, a ready-to-go Fuseki docker container was used[18].

Loading the raw triples yielded a burdensome task. Using the web interface, an attempt to load each of the TTL files was made. However, the web interface only allows to load one file at a time. Moreover, with files of sizes over 50 MBytes, the loader stopped and returned an error message: "GC overhead limit exceeded". Even manually splitting up files into multiple subsets did not help. In fact, the loader got slower as the amount of imported data increased. Possessing approximately 25 raw data files, combined with the increasingly longer loading time of several hours per file, the task was already set to be highly inefficient, and other solutions were investigated.

tdbloader2 is a command line bulk loader and index builder, provided into the Jena framework. It permits the loading of the triples into the database in an automated and much faster way. For example, the loader was able to import a total of approximately 35 million in approximately 6 hours, without yielding any error message regarding the lack of memory space.

[16] https://jena.apache.org/
[17] http://graphdb.ontotext.com/
[18] https://hub.docker.com/r/stain/jena-fuseki/

Querying a dataset of approximately 35 million triples is reasonably fast; at least for simple queries. However, when more complex and complicated requests were executed, it used to block other queries for several minutes or even getting stuck to the point in which a shutdown was necessary. Here, the terms *simple* and *complicated* refer to queries that are respectively very specialized and very vague. An example of both can be seen in listing 3.12 and 3.13.
Note: the prefixes declarations were omitted in both queries.

```
SELECT ?name WHERE {
    # we are looking for the name of a film
    ?film rdf:type dbp-owl:Film.
    ?film dbpprop:name ?name.

    # the film has a director
    ?film dbpprop:director ?director.

    # the director has a name Clint and family name Eastwood
    ?director rdf:type dbp-owl:MovieDirector.
    ?director foaf:name "Clint".
?director foaf:familyName "Eastwood".
}
```

Listing 3.12: Example of a simple sparql query

```
SELECT DISTINCT ?name WHERE {
    # we are looking for some resource x of unknown type xtype
    ?x rdf:type ?xtype.

    # with x having a name
    ?x dbpprop:name ?name.

    # which is somehow connected to some resource y, z and w
    ?x ?py ?y.
    ?x ?pz ?z.
    ?x ?pw ?w.

    # where y has a property releaseDate 2016
    ?y dbpprop:releaseDate 2016 .

    # where z has a property familyName "Eastwood"
    ?z foaf:familyName "Eastwood".

    # where w has a property name "Clint"
    ?pw foaf:name "Clint".
}
```

Listing 3.13: Example of a complex sparql query

Jena was not able to find any result for the complex query in a reasonable amount of time. The execution was stopped after 3 minutes. However, it was able to find a solution to the simple query and display it in less than 1 second. This is a clear hint that the amount of paths that Jena needs to evaluate in the complex query is too large for its capabilities.

GraphDB

GraphDB is a RDF triplestore developed by Ontotext. There exist multiple commercial versions, but a free version was used in this project. A ready-to-go docker container is

provided as well[19].

Motivation for undertaking the experimentation of GraphDB were the limitations in the complexity of queries that Jena is able to execute. Additionally, the assumption that the setup operations (installation and loading) could be run within a 1-2 hours and almost completely autonomously was made. The reasoning behind this estimation came from the fact that GraphDB was proven to be able to load approximately 225'000 statements per second with hardware capabilities similar to those provided by our cluster[17]. Our dataset including 35 million triples could then hypothetically be loaded in less than 3 minutes.

Loading the triples on GraphDB was a surprisingly fast operation. In contrast to Fuseki, multiple files up to 200MB size each can be uploaded in bulk even through the user interface. The results were astonishing: on the same physical machine on which Fuseki is running, the import time for all raw triple files summed up took approximately 7 minutes. At that point, the database was ready to be queried.
The test has been executed on the same physical machine, with the same dataset loaded into both triplestores.

Comparison: Jena Fuseki vs GraphDB

The hypothesis that presenting the same dataset using GraphDB was not going to improve query time performances nor affect the set of executable operations was made.
To prove this evaluation, a complex SPARQL production query with filters and unknown predicates (wild-cards) is considered. Wild-cards were causing troubles with Jena not being able to retrieve any result in a reasonable amount of time. A response time of under 1 second would be acceptable.
The listing 3.13 represents a complex query that is part of an experimental approach tried during the lifetime of this thesis. As described in paragraph *Querying* of subsection *Apache Jena Fuseki*, Jena is not able to answer such request on the dataset. Surprisingly, when running the same query through GraphDB's web interface, the request took 5.1 seconds. This response time is still far from ideal, but the complexity of the query is extreme, especially considering that all subject types and many predicates were omitted. Important to note is also that from the triplestore's perspective, the resources z and w are not necessarily the same, but can be, which further increases the number of possible combinations to be evaluated by the database.

3.3 Lessons learned

R2RML mappings are a very powerful and yet easy to comprehend instrument. The RAM capacity and elaboration time required can make it burdensome to convert relational data into RDF format. However, with the approach taken, mapping templates are decoupled, causing them to be even simpler to understand and modify. Further, if specific relationships require change, just a granular part of the database needs to be re-transformed and re-imported, increasing the speed of the operation.
R2RML Parser turned out to be a viable and efficient tool, despite not completely adhering to the R2RML standard specifications.

[19] https://hub.docker.com/r/ontotext/graphdb/

The introduction of the GraphDB semantic database proved to exceed the expectations, by notably increasing the speed of the queries on the dataset. Additionally, requests that were previously thought to be impossible to answer, were made viable.

The amount of RAM needed for the conversion depends on the amount of triples to be generated in a single mapping, which can be calculated with a simple formula: number of records in the query resultset multiplied by the number of `rr:predicateObjectMap` and `rr:class` blocks. The length of the resultset can be calculated with the SQL operator `COUNT`. While the amount of `rr:predicateObjectMap` blocks is strictly related to each mapping file, the number of `rr:class` statements can be either *0* or *1*.

In the case of our largest mapping, the one of the actors, the query returns a resultset containing *2'286'018* records, while the mapping itself contains *5* predicate-object blocks plus *1* rr:class statement. Based on the formula mentioned before, *2'286'018* multiplied by *6* equals to *13'716'108*, which is the amount of triples produced by the mapping file. Therefore, considering that *128 GB* were enough to generate *13'716'108* triples, we can assume that approximately *107'000* (`13'716'108 / 128`) triples can be generated from each GB of RAM available.

To reduce the amount of triples generated from a single mapping we suggest to divide the predicate-object assignments over multiple files, rather than temporarily modifying the SQL query to limit the amount of records extracted. Justification for this is that the amount of rows in the relational dataset can vary. In such cases, approaches like using the BETWEEN filter on the table's *ID* could stop being effective when changes on the relational data are applied (e.g. many more records are added to the source database). Moreover, the dumping of the mappings and the generation of the respective property files can be automated with a script, while changes on the SQL queries cannot.

For future approaches the use of *GraphDB* and *R2RML Parser*, together with our conversion automation script, is highly advised.

Chapter 4

Infrastructure

In this chapter we discuss the decisions behind our design choices in the sense of its general objective, its components and their purposes.

One of our first idea was to implement a simple rule-based bot and after that running an experiment to gather data from the users. With this new data, it may have been possible to see what and how people talk about movies and thus introduce new intents or modify the existing one. This experiment has been discarded after we realized that quepy has too many limitations and we decided to stop working on it.

4.1 Components

A chatbot consists of multiple parts interacting with each other. It is important to have a solid structure in order to be able to swap components with ease and allowing the introduction of new modules without difficulties. To better understand how to divide the application, we need to identify all the fundamental elements.
First of all, an interface allows the user to communicate with the chatbot. There are two main ways for a user to provide the input: written and spoken - the input methodology should not interfere with the inner workings of the rest of the infrastructure.
Second, a service which understands the user's input and build a query or a textual response - we call this component *the smart machine*.
Third, a database which contains all the information that we want to query.
We added one more main component: a system to log all the conversations and the processed information extracted during a conversation, such as: the intent, the entities and eventually a feedback from the user. The latter could be extremely helpful evaluating the quality of the answering system and gives the possibility to improve the system overall.

Figure 4.1 shows the main components of the bot and the interaction between them.

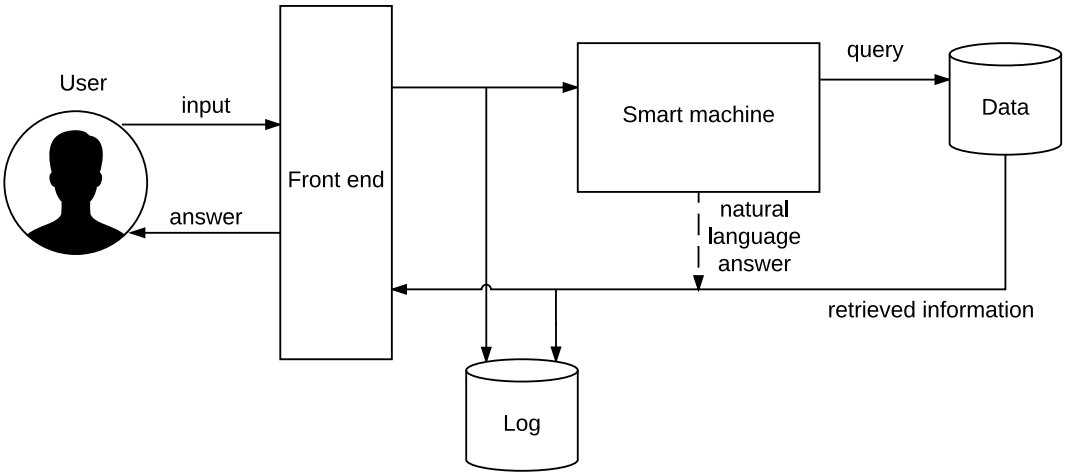


FIGURE 4.1: General infrastructure

4.2 The user interface

The user-interface for a written input can be provided by either a new simple (web) application or by integrating the bot in an already existing messaging app, such as Facebook Messenger, Telegram or Slack (there are many more possibilities which we do not mention but they all work in similar ways).
Using a third-service messaging-app is simple and straightforward but there are some drawbacks which need to be considered. The user has to be registered to the service and all the conversations are recorded by those big companies. Furthermore, relying on a third-party service means that if the service goes down, the bot goes down as well. (We are not saying that our bot cannot go down, but instead, that we do not rely on external services.)

We decided to build our own web interface in order to have more flexibility and to have it offline during the development. Our solution allows a possible future porting to a messaging app effortless.

Our web interface consists of a query endpoint and a dynamic HTML page. Both of them are provided by the open-source framework *Flask*.

4.2.1 Query endpoint

The endpoint to query the chatbot can be started with the command `python server.py` and can be reached with the path `/query?question=`. It returns a JSON document with the answer from the bot. In order to simplify the implementation of this part, the received document is the same as the one stored in the log database. (Check section 4.4 for more information.) When using this endpoint, it is important to encode the characters that are not allowed in a url. For example, the question "Trailer of Inception" will be encoded as `/query?question=Trailer%20of%20Inception`.

Furthermore, to query the endpoint, the user needs a session id. This allows the bot to identify the user and it is later used when retrieving the previous context from the log system. When calling the home page `/` and the session id is not set yet, a new random

id is provided by the server. The given id is currently just a randomized integer from 0 to 10000 and does not check for duplicates. (This can be easily improved but it is not included in the current goal of our thesis.)

The path `/logout` is used to destroy the session id and the user will need to reach the home page again before being able to query the bot again.

4.2.2 Front end

We invested a reasonable amount of effort in the front end in order to make it looks good and appreciable. The graphical front end is implemented in HTML, CSS and Javascript and simulate a chat box, similar to other messaging apps. We used a template[1] based on Bootstrap from Twitter[2].

Figure 4.2 shows an example of a user requesting to see the trailer of a movie. The blue box is the user's input while the light orange is the answer from the bot.

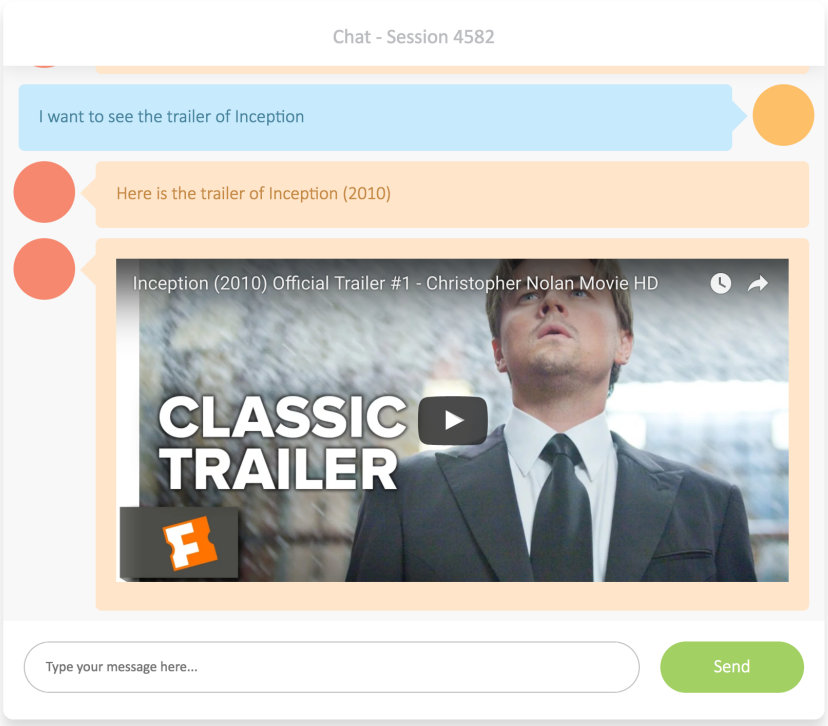


FIGURE 4.2: A user asking the trailer of a movie

With the help of the library CharJS[3] we are able to provide the user with piechart (figure 4.3) and histograms (figure 4.4).

[1] https://bootsnipp.com/snippets/ZlkBn
[2] https://getbootstrap.com/
[3] http://www.chartjs.org/

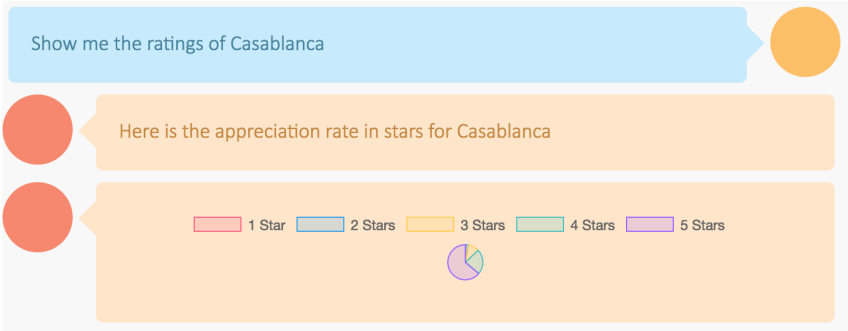


FIGURE 4.3: Piechart in the chat

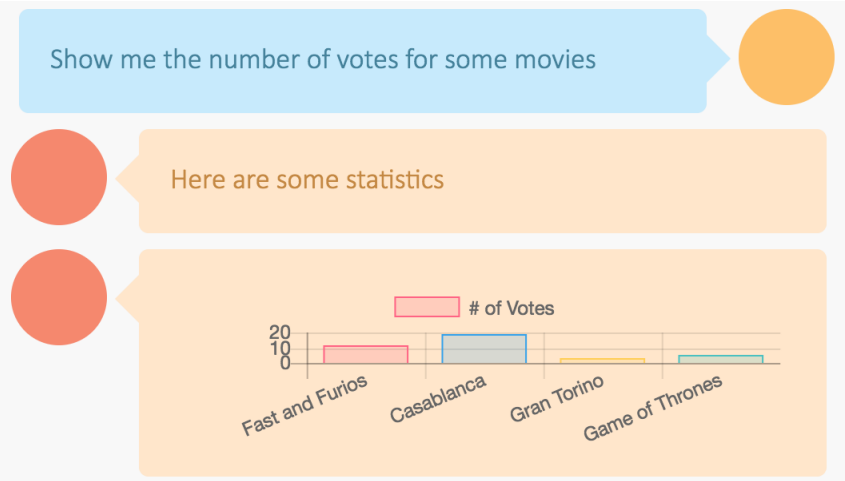


FIGURE 4.4: Histogram in the chat

Similar to textual answers, the actual data to plot is stored in the attachments, together with the type of data. The parser is then able to process the data in the right format.

In the case of the Youtube trailer, two messages have been received and they are parsed in two different way. The data received by the client is shown in listing 4.1. (Some fields have been removed to improve readability.)

The attachments are processed by the Javascript file `parser.js`. This file contains many methods, one for each type of attachment. Listing 4.2 shows how to handle a Youtube attachment. The resulting string is then inserted in a new message box.

```
1  {
2     "answer":{
3        "actor":"bot",
4        "attachments":[
5           {
6              "content":"Here is the trailer of Inception (2010)",
7              "contentType":"text"
8           },
9           {
10             "content":"https://www.youtube.com/embed/YoHD9XEInc0",
11             "contentType":"youtube"
12          }
13       ]
14       "sessionId":4582,
15       "timestamp":"Mon, 29 May 2017 12:20:02 GMT"
```

```
16     }
17 }
```

Listing 4.1: JSON document describing the Youtube trailer

```
1  var parse_youtube = function(content){
2      return '<iframe width="100%" height="315" src="' + content + '"
       frameborder="0" allowfullscreen></iframe>';
3  }
```

Listing 4.2: Processing a Youtube content

4.3 The smart machine

The user's input is processed by the smart machine: it take the input as a textual form and produces either a database-query, a natural-language answer or both. This part consists of multiple independent components where each of them handles a different problem or implements a different technique to provide an answer. For example, one such component can use regular-expression to classify an input; another one may use an online-service to extract entities instead and an additional one could handle movies recommendation.

4.3.1 Redirection

The smart machine needs to know to which component the input should be redirected. Since we wanted to focus more on other parts of the chatbot, we decided to use a simple system to handle the redirection. When the smart machine receives an input, it is forwarded to the first component. If the component is not able to produce a valid output or it does not know how to handle the request, the input is sent to the second component, and so on. Finally, when a component is able to provide an answer, the result is forwarded back to the smart machine which will then send it back to the user.
In addition, the input and the resulting messages are sent to the module `History`, which stores them into the log database.

4.3.2 The commandBot

The first component that we implemented is the *CommandBot*. It is mostly for demo purposes but it can also be used for adding features to the chatbot - we actually use it to handle the user's feedback and the welcome message. It controls all the inputs starting with the colon symbol (`:`) which are specific commands and do not need to be processed further.

The `:hello` command has an important role: it is automatically sent by the client when the web interface is opened and tells the CommandBot that a user wants to talk with the bot. The user will then receive a welcome message together with other information.

The `:feedback` directive is used to manage the user's feedback about the last answer that the bot produced. In order to provide a valid feedback, the message needs to match the following syntax:

```
1 :feedback <satisfaction\%> <expected type of result>
```

(This is just an example on how we implemented it but it can be easily changed, allowing more information to be recorder; we decided to keep it simple at the beginning.) The feedback is then forwarded to the *History* module which will update the corresponding entry (the last message sent by the bot to the user) in the log-database.

The `:help` command returns a list of all the commands which are allowed. The remaining commands are for test-purpose only: `:pie`, `:histogram`, `:list` and `:youtube`. They show how a pie chart, a histogram, a list or a youtube link may look like. (As our main focus was retrieving information, it is not possible to require these types of data.)

4.4 Logging

There are many reasons that came up to introduce this component. These motives vary from keeping track of the performances of the system to handling long-term memory. To better understand the significance of this module, we have to go back in the earlier phases of the project, when we were looking for dialogs about movies. Apart from Facebook, which provides a dataset of dialogs about movies, we did not find any other collection of conversations about the interested topic. We therefore planned to run an alpha-test with an early version of the chatbot in order to comprehend and understand what and how people talk about movies. (We eventually did not run an alpha test, but our solution can easily used for a future work.)

We realized that a log-system could be really useful if implemented in an appropriate way and therefore we needed to find a good solution to our case. At the beginning, we were not completely aware of how our data will look like. Hence we based our assumption on other system, such as API.ai and LUIS. Even though both system aim to the same goal, they use different representation of the processed data. We combined both systems to create our own data representation.

We had to figure out a suitable way to describe the conversations. A conversation consists of multiple messages performed by either a specific user or the chatbot.
Since a conversation should be uniform from both counterparts, the messages are threatened in very similar way. In the normal case, the user sends only one message at the time, while the bot may returns multiple text. Table 4.1 shows an example of the user asking one question and receiving three responses.
In order to have both the messages of the user and the ones of the bot consistent and uniform, each of these independent texts are considered attachments and are grouped together in a single document. Figure 4.2 show how these attachments are stored in a JSON format. (We left some fields out in order to increase the readability.) We are now able to reconstruct a conversation and analyze it. It is worth mentioning the missing field `extra`, which contains additional information about the component which generated the answer, such as: database query, metadata, entities extracted and intent.

User says	Chatbot says
Show me the trailer of Inception	
	Here it is
	`<YOUTUBE TRAILER>`
	Do you want to know more about it?

TABLE 4.1: Example of a user conversating with the chatbot

```
{
  "_id" : ObjectId("xx10"),
  "lang" : "en",
  "sessionId" : 4199,
  "actor" : "user",
  "attachments" : [
     {
       "content" : "Show me the trailer of " +
                   "Inception",
       "contentType" : "question"
     }
  ]
}
```

(A) User says

```
{
  "_id" : ObjectId("xx11"),
  "lang" : "en",
  "sessionId" : 4199,
  "actor" : "bot",
  "attachments" : [
    {
      "content" : "Here it is",
      "contentType" : "text"
    },
    {
      "content" : "[...]/v=YoHD9XEInc0",
      "contentType" : "youtube"
    },
    {
      "content" : "Do you want to know " +
                  "more about the movie?",
      "contentType" : "question"
    }
  ]
}
```

(B) Chatbot say

TABLE 4.2: Example of a user conversating with the chatbot

Chapter 5

Approaches

This chapter gives an overview on the inner-workings of the two libraries Quepy and RasaNLU. We propose a solution to extend the capabilities of Quepy, by allowing the generation of queries with *FILTER* and other modifiers. These features are not integrated in the library.
Further, a technique to understand how words are related to each other by looking at their representation in the RDF store is presented. This allows to associate new sentences to an old context.

5.1 Regular expression using Quepy

Important aspects for comprehending Quepy's functioning are introduced.
Note: Many mechanics are not described precisely in the official documentation of Quepy and a lot of effort was required in order to comprehend the inner-workings. Thus, we document here what we have learned about the library from our experience in order to clarify following chapters.

5.1.1 Question templates

`QuestionTemplates` is one of the core classes of Quepy. Every object extending this class represents a question (an intent) and contains a regular expression written using REfO's syntax. From this point onwards, the term regular expressions refers to regular expression written using REfO. They are defined in the `regex` field of the class and are composed by any combination of the following three structural elements.

Lemma Lemmas (or lemmata) are words considered as their citation form, and are used to express any variation of the form of a word, especially verbs or nouns. For example, if a regular expression were to contain `Lemma('run')`, it will match all the words which are form variations of the lemma *run*; such as *run*, *running* and *runs*.

POS-Tag A Part-Of-Speech Tag represents the grammatical category of a word in a corpus. Such categories include singular nouns (denoted by "NN"), plural nouns ("NNS"), infinite verb tenses ("VB") and many others[1].
POS-Tags can be used in regular expressions by using the notation `POS(<tag>)`. In this

[1]For the complete list of tags, consult the appendix C

case, the regular expression is matched when the word has the same tag `<tag>`.
For example `POS('NN')` matches all the singular nouns while `POS('VB')` all the infinite verbs. Such statements can be combined together in a regular expression in order to match all the nouns followed by a verb. The result is the following: `POS('NN') + POS('VB')`.

Particle A particle is used to define frequent patterns in the form of regular expression. Additionally, it defines the interpretation of the matched text using the relations defined in the Domain Specific Language. Particles are most often used inside regular expressions of a *QuestionTemplate* subclass, but they can be nested in other particles as well.

5.1.2 DSL

The Domain Specific Language defines the relationships between subject and object in the domain, which are later used to build SPARQL queries.
DSL are created by extending one of the following classes.

FixedType defines the type of a resource. A resource is a subject or object identified by an IRI. For example, people have a relation `rdf:type` with the value `foaf:Person`. The listing 5.14 shows how the person type is defined.

```
1 class IsPerson(FixedType):
2     fixedtype = "foaf:Person"
```

Listing 5.1: Fixed type relation

FixedDataRelation defines a relation between a resource and a literal (a constant), where the latter can be either a string or a number.

For example, all the resources having a name could be filtered by using the class `HasName()`, which is implemented in listing 5.2.

```
1 class HasName(FixedDataRelation):
2     relation = "dbpprop:name"
3     reverse = True
```

Listing 5.2: Fixed data type relation

FixedRelation defines a relation between two resources. For instance, an actor who stars in a movie is pointed by the movie itself through the predicate "dbpprop:starring". The relation can be described with `StarsIn` as shown in listing 5.3.

```
1 class StarsIn(FixedRelation):
2     relation = "dbpprop:starring"
3     reverse = True
```

Listing 5.3: Fixed data type relation

The variable `reverse` is used in FixedDataRelation and FixedRelation to determine the direction in which the predicate has to be interpreted. This depends on how the RDF is modeled and from the subject on which we want to apply a statement. Assume that a

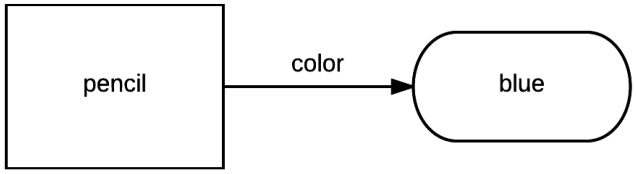


FIGURE 5.1: RDF Graph containing information about a pencil

RDF graph models a pencil resource which has a property color pointing to the constant string value blue, as shown in figure 5.1.

If a verbal statement had to be made about the pencil, one would say that the pencil has color blue. In this case, the pencil is the subject of our statement and blue is the object. This statement is defined in the listing 5.4

```
1 class HasColor(FixedDataRelation):
2     relation = "color"
```

LISTING 5.4: Statement "Pencil has color" programmatically

On the other hand, if the statement were made from the perspective of the color itself, one would affirm that blue is the color of the pencil. The subject is now blue, while the object is the pencil. That means that the natural roles have been reversed, and therefore the direction of the relation needs to be reversed.

Relations algebra

DSL statements are combined together and used to generate SPARQL queries. An example to better understand this mechanic is provided.
`IsPerson()` initializes a resource (say A) of type Person. `HasName('Mike')` defines another, independent resource (say B) with the property `dbpprop:name` equal to `Mike`. Listing 5.5 represents the result of such independent operations represented as triple statements.

```
1 A = ?x rdf:type dbp-owl:Person
2 B = ?x dbpprop:name "Mike"
```

LISTING 5.5: Resource A is a Person

Adding the two objects together, `IsPerson() + HasName('Mike')`, results in a definition of a new resource C of type person and named Mike, as represented by listing 5.6.

```
1 C = A + B = ?x rdf:type dbp-owl:Person
2             ?x dbpprop:name "Mike"
```

LISTING 5.6: A plus B joins the heads of the statements

Further, to extract the family name, a `FixedDataRelation` as shown in listing 5.7 is used, together with the result of `FamilyNameOf(IsPerson() + HasName('Mike'))`.

```
class FamilyNameOf(FixedRelation):
    relation = "dbpprop:familyName"
    reverse = True
```

Listing 5.7: Family name of as subclass of FixedRelation

The result of wrapping `FamilyNameOf` around resource C is shown in listing 5.8

```
C = ?x rdf:type dbp-owl:Person
    ?x dbpprop:name "Mike"
    ?x dbpprop:familyName ?y.
```

Listing 5.8: Resource C after being wrapped into FamilyNameOf

5.1.3 Pipeline

The three steps to process the raw user's input and generate a SPARQL query are the following[18]:

- Parsing
- Matching and intermediate representation
- Generate SPARQL query

Parsing

This component receives an input text and with the help of the NLTK POS-Tagger, associates each word to the corresponding lemma and POS-Tag.

The output has the following structure:

```
Word1|Lemma1|POS-Tag1 Word2|Lemma2|POS-Tag2 Word3| etc...
```

For instance, the word "running" will be converted into "running|run|VBG".

Example of sentence parsing Suppose we want to create a template to match questions asking for the release date of a movie.
Requisites: A question template, able to match to the given input.

```
Lemma(when) + Lemma(be) + Particle(Movie) + Lemma(release)
```

Input: "When was Gran Torino released"
Output:

```
When|When|WRB was|be|VBD Gran|Gran|NNP Torino|Torino|NNP released|release|
    VBD
```

In this case, the words "Gran" and "Torino" represent the values matched by the movie particle.

Match and intermediate representation

The output of the parsing phase is used as input to match against a regular expression. The matching patterns are then extracted in a programmatic representation that can be graphically visualized and can be used to generate SPARQL queries.
An example is provided to better understand the role of particles and question templates.

Full template question and particle example In this example, we try to match and build a query to retrieve the release date of a movie by answering the following question:

When was Gran Torino released?

This case is implemented using one particle and one question template as shown in listing 5.9 and 5.10. (Notice, this is a simplified version of the actual version.)

```
1 class MovieReleaseDateQuestion(QuestionTemplate):
2     # example of mateches: When was Gran Torino released?
3     regex = Lemmas("when be") + Movie() + Lemma("release") + Question(Pos(
      ".."))
4
5     def interpret(self, match):
6         release_date = ReleaseDateOf(match.movie)
7         return release_date
```

Listing 5.9: Question template example

```
1 class Movie(Particle):
2     # the name of the movie can be composed by multiple nouns and
      adjectives
3     regex = Plus(Pos("NN") | Pos("NNP") | Pos("JJ"))
4
5     def interpret(self, match):
6         # the matched tokens represent the name of the movie
7         extracted_name = match.words.tokens
8         return IsMovie() + HasName(extracted_name)
```

Listing 5.10: Particle example

The parsed user's input is matched against all the question templates. The regular expression of the question template makes use of the particle movie. Thus, a part of the sentence is first interpreted by the particle and then by the question template.

Particle interpretation Part of a sentence is matched agains the regular expression. In case of a positive response, the matched results are passed into the `interpret` method, which defines what to do with the extracted information. In this case, the matched result is the name of the movie and it is retrieved using `match.words.tokens`.

The next part consists in building the data representation of the extracted information. DSLs are used to define the type of resource and the parameters to select. By using `IsMovie() + HasName(extracted_name)`, we are basically saying that *the matched result represents a resource of type* movie *and it has a* name *equal to the extracted one* .
To be more clear, `IsMovie()` defines a `FixedType` relation and creates a resource with `rdf:type` equal to `dbpedia-owl:Film`. On the other hand, `HasName(extracted_name)` defines a resource with a predicate `dbpprop:name` equal to the literal `extracted_name`.
By adding these two separated resources together (shown in listing 5.11), a new one with

both information is created (shown in 5.12). (Notice that the two resources in the first example x0 and x1 are merged together in x0.)

```
1 x0 rdf:type dbpedia-owl:Film
2 x1 dbpprop:name "Gran Torino"
```

Listing 5.11: Separated resources

```
1 x0 rdf:type dbpedia-owl:Film
2 x0 dbpprop:name "Gran Torino"
```

Listing 5.12: Merged resources

It can also be expressed graphically as shown in figure 5.2.

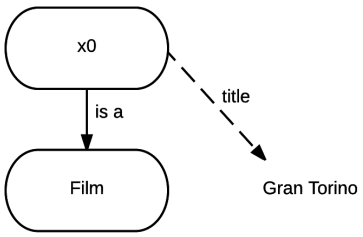


Figure 5.2: Intermediate representation after processing the regular expression of the Movie Particle

Question template interpretation The complete sentence pass the same process as the particle. The `interpret` method of this class defines, similarly to the particle, what to do with the extracted information.
In this case, `match.movie` is the resource which has been processed by the particle. `ReleaseDateOf(match.movie)` is a `FixedRelation` and denote to add a new relation `dbpedia-owl:releaseDate` with an unknown variable to the movie resource.

Listing 5.13 shows the resulting data. (x1) is the unknown variable added by the interpret method of the question template. Figure 5.3 shows the same data in a graphical way.

```
1 x0 rdf:type dbpedia-owl:Film
2 x0 dbpprop:name "Gran Torino"
3 x0 dbpedia-owl:releaseDate x1
```

Listing 5.13: Merged resources

Generating SPARQL queries

Quepy generates SPARQL queries by formatting the set of DSL statements produced in the various `interpret` methods. In this thesis, the mentioned set is referred as *expressions* or *expression list*. The *expression list* contains a series of one or more nodes (equivalent to resources), each containing one or more edges to other nodes.

Note: One should keep in mind that the job of DSL is to set up a python representation of the query, i.e. of the relations between resources based on the ontology provided.

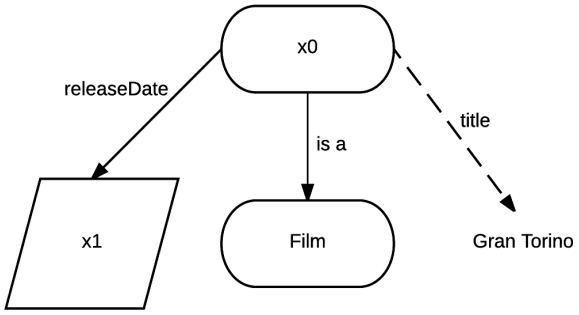


FIGURE 5.3: Intermediate representation after processing the regular expression of the Question Template

The procedure for the conversion from *expressions* to SPARQL query is shown as pseudo-code in listing 5.14[2].

```
1  def expression_to_sparql
2    let e = expressions
3    let prefixes = listOfPrefixes
4
5    query_template =
6      {prefixes}
7      SELECT DISTINCT {target_columns} WHERE {
8        {triples}
9      }
10
11   target_columns = e.head
12
13   for each node in expressions
14     for each edge in node
15       triples.append(adapt(node) + edge.relation + adapt(edge.destination)
      )
```

LISTING 5.14: Pseudo-code for SPARQL generation from DSL's *expression list*

Generating SQL queries in Quepy

In its official distribution, Quepy is not able to generate SQL queries, and a suggested alternative is to use a SPARQL-to-SQL query translator (see [19]). Contrarily to the suggestion, a file responsible for the generation of SQL queries based on *expressions* was created. In the file, not just the *query_template* but also the parsing procedure was modified. Consequently, the code that generated the DSL statements (i.e. the method `interpret` in the `QuestionTemplate` subclasses) had to be adapted. This design decision implies that the necessary *tables*, *where statements*[3] and *join operators* need to be hard-coded in the `interpret` method. The issue mentioned causes the code burdensome to understand, modify or write. No example is provided here, as it would be too challenging to comprehend or explain in brief terms.

[2] https://github.com/machinalis/quepy/blob/develop/quepy/sparql_generation.py
[3] an SQL statement appearing after the *WHERE* clause in a SQL query

5.1.4 Extending Quepy with SPARQL modifiers

The experimentations with Quepy have proved that the queries that it was able to generate were not suitable for a complete solution. The problem lies in the quantity of data contained in the RDF graph and in the lack of support for FILTERS and ORDER BY clauses in Quepy.

To be more precise, the IMDb contains hundreds of thousands of movies. Many of them are unpopular and may barely interest users. In order to provide more pertinent results, sorting the movies by release year or by rating is a necessity.
In addition, SPARQL queries filter the given literals only if their value matches perfectly with the stored one, making a case-sensitive comparison for strings and a type equality test for numbers. If a single character is different, the query will not return any result.

SPARQL's FILTER clause permits to overcome this limitation by allowing the creation of more complex queries which are not restricted to value equality tests. For example, numbers can be matched within a range of valid values or strings can be tested against regular expressions. The latter, enable to ask tests with a behavior similar to SQL's "LIKE" clause.
These features can be very convenient, especially when it is expected that the user will not always provide an exact value.

Support for FILTERs

Consider a simplified version of the question template `PlotOfMovie`, presented in listing 5.15.

```
1 class PlotOfMovie(QuestionTemplate):
2     """
3     e.g.: "what is the Matrix about?""
4     """
5
6     regex = Lemmas("what be") + Movie() + Lemma("about") +  Pos(".")
7
8     def interpret(self, match):
9         return PlotOf(match.movie)
```

Listing 5.15: Simplified PlotOfMovie question template

When a sentence matches against the specified REfO regular expression, the methods `interpret` of the particle `Movie` and the question template `PlotOfMovie` are called in succession. Figure 5.4 displays the generated representation of a matched sentence, as described in detail in section 5.1.3. The corresponding SPARQL query is presented in listing 5.16.

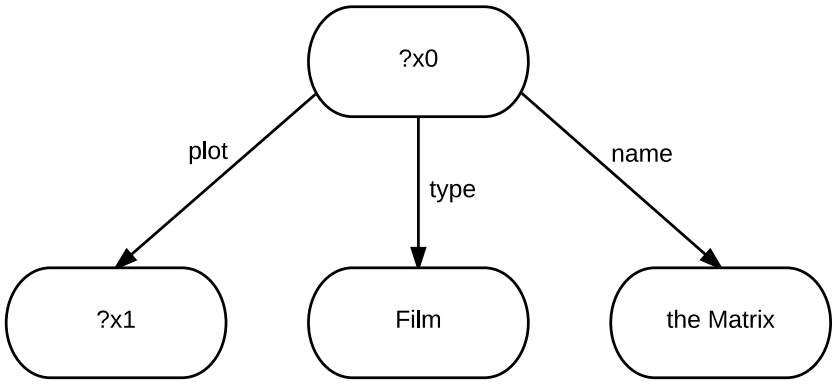


FIGURE 5.4: Graphical representation of the elements extracted from the first sentence

```
SELECT ?x1 WHERE {
        ?x0 rdf:type dpb-owl:Film .
        ?x0 dbpprop:title "the Matrix" .
        ?x0 dbpprop:plot ?x1
}
```

LISTING 5.16: SPARQL query to retrieve the plot of the movie "The Matrix"

The previous literal "the Matrix" of the property `name` has been swapped with the variable `?title`. In addition, a FILTER clause has been added to the query. The filter makes uses of a regular expression to match all resources for which the name contain "the matrix" case-insensitive.
As a result, all the movies whose title contains "the matrix" are matched. The extended method `interpret` is shown in listing 5.17.

To create the FILTER expression, the given film name has to be retrieved. This is done by the function `replace_prop` which replaces and returns the previous value, in this case "the matrix", with the variable `?title`. The clause is then encapsulated in a `MyExtra` object and added to the data.
The second parameter of the function `add_data` is expected to be either a literal or another entity and represents the object of a triple. The extra object is not parsed like the other properties, therefore an empty value can be used.

```
def interpret(self, match):
    plot = PlotOf(match.movie)

    old_value = replace_prop(plot, u"dbpprop:name", u"?name")

    extra = MyExtra('FILTER regex(?name, "%s", "i").' % old_value)
    plot.add_data(extra,u"")
    return plot
```

LISTING 5.17: Interpret method with FILTER clause

Quepy does not handle additional modifiers, therefore the function `expression_to_sql`, which transforms the data into a SPARQL query, has been rewritten. The function is then capable of handling `MyExtra` objects and they are written correctly into the SPARQL query.

Support for ORDER BYs

The ORDER BY clause can be useful to provide the most interesting results to the user first. For instance, if the user requests a list of movies released in 2010, a simple query (shown in listing 5.18) may return some movies which are unpopular, since they are not ordered by the highest rating or number of views.

```
SELECT ?name WHERE {
    ?film rdf:type dbp-owl:Film;
        dbpprop:name ?name;
        ns:release ?release.
    ?release rdf:type ns:Release;
        dbpprop:releaseLocation "USA";
        dbpprop:releaseDate 2010.
}
```

Listing 5.18: SPARQL query without ORDER BY

On the other hand, listing 5.19 makes use of ORDER BY and the selected movies are sorted by descending number of votes received.

```
SELECT ?name WHERE {
    ?film rdf:type dbp-owl:Film;
        dbpprop:name ?name;
        dbpprop:counter ?counter;
        ns:release ?release.
    ?release rdf:type ns:Release;
        dbpprop:releaseLocation "USA";
        dbpprop:releaseDate 2010.
} ORDER BY DESC(xsd:integer(?counter))
```

Listing 5.19: SPARQL query with ORDER BY

The clause ORDER BY is not handled by Quepy, therefore, an additional modification is required.
Similarly to the FILTER clause, the class `OrderBy` has been created and can be used in the `interpret` methods. In listing 5.20 two more items are added: the property counter and the `OrderBy` object which will then be added to the SPARQL query to sort the results by the number of votes.

```
class FilmsOfYearQuestion(QuestionTemplate):
    """
    Ex: "show me movies of 2010"
    """

    regex = Lemmas("show me some movies of") + Question(Lemma("year")) +
    Release()

    def interpret(self, match):
        movie = IsMovie() + Released(match.release)
        movie.add_data(u"dbpprop:counter", u"?counter")
        movie.add_data(OrderBy("DESC(xsd:integer(?counter))"),u"")
        return movie
```

Listing 5.20: Question template with ORDER BY clause

Evaulation

We provide a simple hack to overcome some of the limitations of Quepy. As we decided to move to other alternatives, this modification is still to be considered a prototype, rather than a workable and tested solution.

5.1.5 Conclusions about Quepy

The underlying idea behind Quepy's approach is interesting. In particular, taking advantage of REfO to write regular expressions facilitates the procedure of defining question templates. However, this backfires as development progresses, since writing regular expressions that cover many formulations and rewordings is difficult. Further, with the increasing amount of question templates, regular expressions tend to overlap with each other, especially when they are constructed with the purpose of matching generic phrasings. Additionally, detecting the overlapping regular expressions is extremely difficult.

5.2 Rule based using Rasa NLU

Rasa NLU is a tool written in Python and provides a system to train models for the different tasks. A model is an artifact generated by an algorithm, and it is used to make predictions. Once a model is created, it can be loaded into the tool, which provides two alternative access methods to the model: directly from code or via a HTTP request. A call to the HTTP endpoint, which we opted for, returns a JSON document containing all the information that the model was able to extract from the input text.

5.2.1 Pipeline

The inner-workings of Rasa NLU are simple. Raw input enters the system and gets processed through various modules. The framework contains multiple pipelines with a series of default components, which are arranged in the following order: tokenization (belongs to pre-processing), entity extraction, intent classifier. Per default, Rasa integrates two tokenizer and two entity extraction modules: Spacy[4] and MIT Information Extraction (MITIE)[5]. The intents are then classified by sklearn[6], a popular python library which implements various machine learning algorithms. Additionally, MITIE also offers an intent classifier. The exact inner-working of the components varies from pipeline to pipeline, but the general idea is very similar among all of them

5.2.2 Training

In order to make new predictions, a model needs to be trained. For this purpose, a training set (i.e. a collection of labelled examples) is required. A training example consists of one text, one intent and multiple entities. For instance, the training example *Show me some movies starring Clint Eastwood* is described from the JSON document represented in listing 5.21.

[4]https://spacy.io/
[5]https://github.com/mit-nlp/MITIE
[6]http://scikit-learn.org/stable/

```
{
  "text": "Show me some movies starring Clint Eastwood",
  "intent": "moviesOfActor",
  "entities": [
    {
      "value": "Clint",
      "entity": "foaf:name",
      "start": 18,
      "end": 23
    },
    {
      "value": "Eastwood",
      "entity": "foaf:familyName",
      "start": 24,
      "end": 32
    }
  ]
}
```

Listing 5.21: Training example for Rasa

Since manually writing JSON documents can be a tedious process, a graphical interface is available[7].

5.2.3 How we use it

We mainly use Rasa NLU to overcome the limitations of Quepy. Rasa NLU is much more flexible since it does not rely on regular expressions, which proved to be difficult to write and overlapping with each other (see subsection 5.1.5).

For our prototype, nine sample intents have been created and trainer with a few examples each.

Note: keep in mind that this is just a prototype; additional intents can be easily inserted.

5.2.4 How to use it

Note: Refer to appendix D for instructions on how to set up Rasa NLU, how to train a model and how to run the server.

We trained three models using the pipelines `mitie_sklearn`, `mitie` and `spacy_sklearn`[20]. The same training data can be used for different pipelines, without the need of manipulating it.

The server can be queried with a single HTTP POST request, as shown in listing 5.22. The result of the curl command can be seen in listing 5.23.

```
curl -X POST http://localhost:5050/parse -d '{"q": "Who is the director of
    Ghostbusters?"}'
```

Listing 5.22: Rasa NLU POST request

[7] https://github.com/RasaHQ/rasa-nlu-trainer

```
{
  "entities": [
    {
      "entity": "dbp-owl:Film",
      "value": "Ghostbusters",
      "start": 18,
      "end": 29,
      "extractor": "ner_mitie"
    }
  ],
  "intent": {
    "confidence": 0.8409657898131153,
    "name": "releaseDateOf"
  },
  "text": "In which year did Ghostbusters come out?"
}
```

Listing 5.23: Rasa NLU POST answer

Note: `sklearn` provides also all the others possible intents with their confidence.

5.2.5 Evaluation

When more examples are used in the training phase, the intent classifier and the entity extractor will probably provide better results. Since the focus of this thesis is something else, we did not want to invest too much time into writing manually examples. Therefore, a precise scientific evaluation cannot directly be made. Nevertheless, we noticed significant better results using `mitie`.

5.3 Context creation via default intents

A context consists of a set of resources and literals connected with each other. This representation is then used to generate SPARQL queries.

The initial context of a conversation is created with the help of predefined intents. As a consequence, the first message sent to the chatbot needs to be correctly classified by Rasa.
Currently, nine intents have been implemented in the prototype.

5.3.1 Background

The classes concerning this implementation can be found in the folder `backend/contextbased`[8].
`graph_metadata.py` initializes objects representing the metadata of the graph database, as shown in appendix B. For example, the method `film_resource_descriptor()` returns a `ResourceDescriptor` object which describes the relationships of the type Film within the graph.
`ResourceDescriptor` objects include: the type of the resource (for example `dbp-owl:Film`), a list of available predicates connected to the resource, and a list of

[8] only the most relevant classes are mentioned

default predicates. The latter defines the typical set of values requested by a user to describe the resource, such as the title (in case of a movie) or the full name (in case of a person).
A predicate is defined using a `PredicateDescriptor` object and contains: the domain of the predicate (such as `dbp-owl:Film`), the predicate (for example `dbpprop:plot`), and the range (often `xsd:string`).

Note: The generation of the metadata could have been automated by executing a special SPARQL query, that would request all kind of resource and literal types. This query necessitates approximately 4 minutes to execute and return a result. However, the metadata would have had to be parsed correctly, and we wanted to avoid losing too much time on this process. Therefore, the decision to utilize a hard-coded definition was made.

As a result, the complete graph schema can be described using the previously mentioned objects. This is used to deduce possible relationships between multiple resources or between resources and literals. Our context-recall approach, as well as many of the solutions proposed to deal with missing information, relies heavily on this data.

5.3.2 How to create a context

An example provides a better understanding on the manner in which a context for a specific intent is created.

The user's request "I would like to watch a movie of Quentin Tarantino" is correctly classified by Rasa as intention moviesOfDirector, together with the two entities `foaf:name` (Quentin) and `foaf:familyName` (Tarantino).
The method which handles the predefined intent `moviesOfDirector` is called and the entities are passed as parameters. This method creates the resources and their connections to the entities, which, together with the intent's target[9], represent all information necessary to build a SPARQL query.

Note: the target information is stored directly in the representation of the resources, in form of a predicate with no object information (object = `None`).

The steps for implementing the default intent `moviesOfDirector` are the following, together with the code in listing 5.24:

1. The two resources representing the movie and the director are created
2. The values of the entities extracted by Rasa are retrieved: `foaf:name` and `foaf:familyName`
3. The properties representing `foaf:name` and `foaf:familyName` are created and added to the director resource, together with their values
4. The predicate describing the relationship between a movie and the director is inserted
5. The predicate `name` of a movie is added with the value None, defining it as the target of the query

```
1 def moviesOfDirector(result_entities):
2     # 1. create two new resources
3     movie_resource = Resource(film_resource_descriptor(), 1)
4     director_resource = Resource(director_resource_descriptor(), 1)
```

[9] intent's target refers to the value of a predicate that the user would like to know

```
5
6       # 2. retrieve values from the extracted entities
7       director_name = result_entities.value_of('foaf:name')
8       director_familyname = result_entities.value_of('foaf:familyName')
9
10      # 3. add the filter on the name
11      name_predicate = PredicateDescriptor('MovieDirector', 'foaf', 'name',
        '', 'string')
12      familyname_predicate = PredicateDescriptor('MovieDirector', 'foaf', '
        familyName', '', 'string')
13      director_resource.add_spo(name_predicate, director_name)
14      director_resource.add_spo(familyname_predicate, director_familyname)
15
16      # 4. add a relationship between the movie and the director
17      director_predicate = PredicateDescriptor('Film', 'dbpprop', 'director'
        , 'dbp-owl', MovieDirector)
18      movie_resource.add_spo(director_predicate, director_resource)
19
20      # 5. define the target predicate for the movie resource
21      title_predicate = PredicateDescriptor('Film', 'dbpprop', 'name', '', '
        string')
22      movie_resource.add_spo(title_predicate, None)
23
24      return [movie_resource, director_resource]
```

Listing 5.24: Context creation of the intent moviesOfDirector

5.3.3 Generating the SPARQL query

The set of resources (and their relationships) created by either a default intent or a context-recalling technique is passed to the method `generate_sparql_from_context()`. All the resources have a list of properties which have been assigned to them, including their values. Thus, generating a SPARQL query consists in assigning the resources variable names, and simply writing to text simple triple statements.

5.4 Context-recalling

This chapter explains and analyses the procedure adopted to recall the context of a discussion. Further solutions to overcome the encountered obstacles of our approach are explained and analyzed. Not all of the solutions presented have been implemented, but they are to be considered as suggestion and as guidelines for future implementations.

5.4.1 Our definition of context

In his work "Understanding and Using Context", Anind Dey provides a definition of context which best approximates ours:
"Context is any information that can be used to characterise the situation of an entity" [21].
For us, context is a set consisting of multiple resources connected to each other. In other words, this is the equivalent of a subset of a graph. This concept builds the basis of our idea and is applied throughout this chapter.

5.4.2 Our definition of entity

In this thesis, the word "entity" refers to a predicate which possesses a value, i.e. a literal or a resource.
It is important to understand the difference from a resource, which is typically the value of a subject or object.

5.4.3 Introduction and generic problems

As part of our research, we analyze a technique to handle sentences which relate to the currently discussed topic (i.e. the context). The main idea lies in using a rule-based framework to construct a starting context. Further, we apply our proposed solution for sentences which relate to the context, and have not been mapped onto an intent from the rule-based framework.
To be more precise, Rasa NLU is used on all input sentences, but builds a meaningful context only when they are mapped with high confidence to an intent, such as: "Show the movies of 2016" or "Who is the director of Titanic?". These kind of phrasings are included as training data into Rasa's model. Our solution comes into play when Rasa NLU does not recognize the intent with high confidence or does not fully extract the entities from the text. This usually occurs when the framework has not been trained on a specific sentence. In this case, we apply our technique by making use of the information that Rasa NLU was able to extract, together with the context that was build in the past. Our proposal could be implemented in other rule-based solutions, such as Quepy, which was used earlier in the project. However, we decided to introduce our own proof of concept.

5.4.4 A granular view on our reasoning chain

Please consider the following granular view of our reasoning: the final objective besides context recalling is to map the input onto a SPARQL query, which retrieves data from a RDF graph. There, information is stored in form of triples, which are composed by 3 elements: subject, predicate and object. By considering and visualizing only the endpoints of this virtual chain, it is noticeable that our objective is to match an input to a set of triples in the RDF graph.
To conclude this reasoning chain, our final, formal objective, is to "identify triple statements in the input sentence", out of which we are able to construct a SPARQL query.

5.4.5 The 4 fundamental elements for a successful query generation

In order to generate meaningful and complete SPARQL queries, a statement must contain the following elements:

- Subjects
- Predicates
- Objects
- Target predicate(s)

The amount of subjects, predicates and objects, excluding the target predicate, must be equal. This way only complete triples can be formed.
An ideal case would be having the user writing sentences in form of triples, as it would be fairly easy to map the input onto a query. In fact, the query would be almost written by the user himself. Additionally, the user could specify the target information, e.g. by leaving the value of a statement to unknown.

Example *Movie title ?x. Movie has a Director. Director has name "Clint". Director has name "Eastwood".* In this case, there are many simple sentences and each of them follows a precise and concise structure. This structure equals to the one in which triples are defined. Thus, generating valid and meaningful queries will not be challenging. A system that makes use of special keywords and provides instructions to the user such as a list of commands, types and relationships, would be the equivalent of an interface for querying RDF endpoints. However, we cannot expect a user to be willing to insert so much information himself, and in such a tedious fashion.

5.4.6 The idea in an example

The example clarifies the general idea about the proposed context recalling solution. Suppose that a user asks for a simple list of movies in his first message, and then specifies an additional filter in the second one.

Show me the movies of 2016

The first sentence is correctly labeled with the intent `moviesOfYear` and with high confidence by the Rasa module. In addition, the entity `dbpprop:year` (which in this case happens to be a predicate) pointing to the constant integer value *2016* is extracted. Since the intent is pre-defined and is identified with high enough confidence, we create a basic context from scratch. The creation of context regarding default intents is hard-coded.

Figure 5.5 shows a graphical representation of the query containing the extracted elements while listing 5.25 shows the actual SPARQL query.

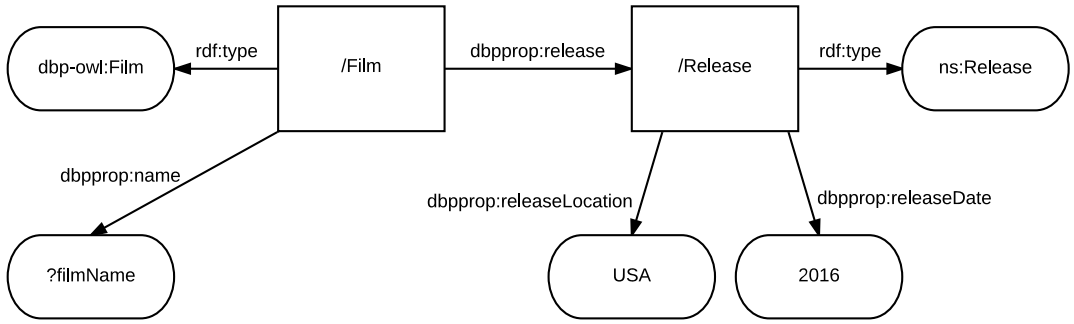


FIGURE 5.5: Graphical representation of the elements extracted from the first sentence

```
1  select ?filmName where {
2      ?film rdf:type dbpediaowl:Film;
3            dbpprop:name ?filmName;
4            dbpprop:release ?release.
5      ?release rdf:type ns:Release;
6            dbpprop:releaseDate "2016";
```

```
7               dbpprop:releaseLocation  "USA".
8   }
```

Listing 5.25: SPARQL query from intent *moviesOfYear*

In his second message, the user says:

Only the ones starring Tom Cruise

Rasa NLU is in this case not very confident about the classified intent but it does manage to extract and identify Tom as the value of the predicate *name*, and Cruise as the value of the predicate *family name*. Now that we have Tom as a name and Cruise as a family name, we have to determine which relation they have between themselves and the previous context (in this case, the list of movies of 2016). In the graph's metadata, we look for classes which have the predicate `foaf:name` or `foaf:familyName`. In this case, the classes *Person*, *Actor*, *Producer*, *Writer*, *MusicComposer* and *MovieDirector*. We refer to these types as candidate subjects. From these options we can guess the type of resource that the user has implicitly provided.

Assume that, thanks to a provided function, we were able to determine the right subject for this situation. Tom Cruise is of type *Actor* and we can build the query visualized in figure 5.6 and displayed in listing 5.26.

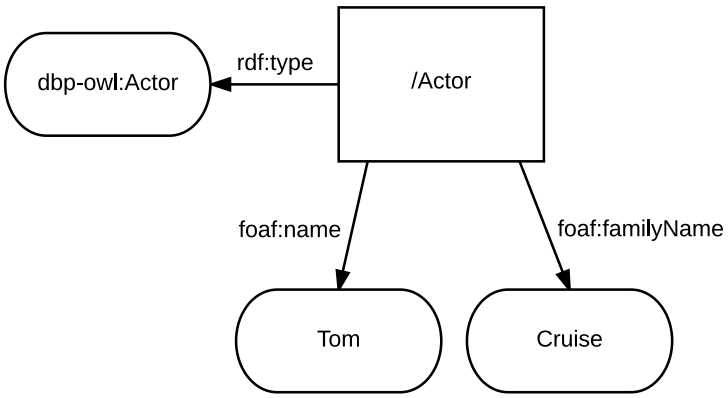


Figure 5.6: Graphical representation of the elements extracted form the second sentence

```
1   ?actor rdf:type dbp-owl:Actor;
2          foaf:name "Tom";
3          foaf:familyName "Cruise".
```

Listing 5.26: SPARQL query derived from sentence *Only the ones starring Tom Cruise*

At this stage, we are left with the context of the previous message (i.e. the "old context") and the one of the current message (i.e. the "new context"). Given these two items, the objective is to find a connection between resources from the old and resources from the new: this operation is called *merge*. In the old context, two resources are present: *Film* and *Release*. However, only the *Film* resource has a possible relation with the resource *Actor*, that is, via the predicate *starring* (see Appendix B). Therefore a connection exists and it is applied.

Figure 5.7 shows the merged context, while listing 5.27 contains the resulting generated query.

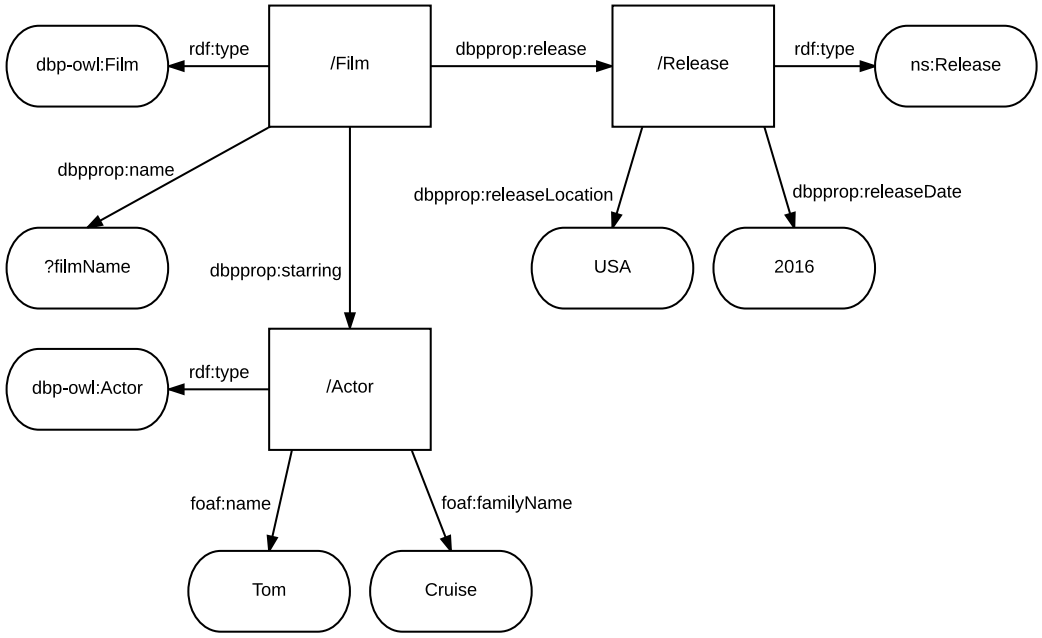


FIGURE 5.7: Graphical representation of the merged context

```
1   select ?filmName where {
2     ?film rdf:type dbp-owl:Film;
3           dbpprop:name ?filmName;
4           dbpprop:release ?release;
5           dbpprop:starring ?actor.
6     ?release rdf:type ns:Release;
7           dbpprop:releaseDate "2016";
8           dbpprop:releaseLocation  "USA".
9     ?actor rdf:type dbp-owl:Actor;
10           foaf:name "Tom";
11           foaf:familyName "Cruise".
12   }
```

LISTING 5.27: SPARQL query deriving from the merged context *Only the ones starring Tom Cruise*

The final result is the one expected: the contexts deriving from the 2 sentences have been merged together correctly. However some problematic aspects that appear in practice influence the feasibility of such an approach. In the next sections, a thorough explanation of the issues occurring when applying this solution to the real world is presented. Additionally, various approaches to (partially) solve the problematics are proposed, out of which a few are implemented.

5.4.7 Implementation of the idea

The underlying idea behind the method is to take advantage of the finite and well-defined connections between resources in our RDF dataset.
Properties belong to a specific set of resource types, have a distinct name and point to a finite set of resource or literal types.
This characteristic is called *graph metadata*, or simply *metadata*, and is essential for the functioning of our context-recalling approach.

Keep in mind that the context is a composite of resources, containing connections to literals and other resources through predicates. If Rasa is able to classify the intent of a sentence with high enough confidence (above 60%), it creates a new hard-coded context, referred as *new context*. Otherwise, it is only able to extract entities out of the sentence. We assume that when the tool cannot recognize the intent, the user has inserted a sentence which refers to the previous context. Therefore, the idea is to connect each entity which Rasa extracted from the sentence with the resources of the old context, leveraging on the known connections between them.

First, a subject (a resource in this case) has to be assigned to each of the extracted entities, since Rasa did not create a hard-coded context. For this, we verify which type of subject (or resource) possesses a connection matching with the entity's name and its type of value.
For example, the entity `dbpprop:name` with the value "Titanic", would match with the subject type Film, because it is the only one possessing the `dbpprop:name` predicate. When a suitable subject type is found, a subject is created from it, and a relationship between the newly created subject and the entity is added. Afterwards, the subject and its references are added to a list, which will compose the *new context*.

The next step is merging the new with and old context together. This is achieved by looking in the metadata for relationships between their respective resources. If a connection exists, it is added. After this has been done for all combinations of resources from both sides, the operation is considered complete.

Note: The input context (`last_context`) can be None.

The method, to work, requires the following input elements:

- text
- old_context or None

The output, consist of the newly generated context:

- new_context

```
1  intent, intent_confidence, entities = rasa.extract_info(text)
2
3  # confidence of 60%
4  if intent_confidence > 0.6:
5    new_context = create_hardcoded_context(entities)
6    return new_context
7
8  else:
9    # create context with all the entities extracted from the question
10   subject_list = []
11
12   # parse the entities matched by rasa
13   for each entity in entities:
14     # note: predicate contains attribute name and value
15     type_candidates = graph_metadata.find_type_candidates(entity)
16
17     # make a guess: most appropriate type for the given predicate
18     best_type = determine_best_type(type_candidates, predicate)
19
20     # create a subject from type
21     subject = create_subject_from(best_type)
22
```

```
23      # between the subject and the predicate and the value extracted by
        rasa
24      subject.add_relationship(entity.predicate_name, entity.predicate_value
        )
25
26      # add the created owner to the list of subjects
27      subject_list.push(subject)
28
29  # a context is just a list of subjects with their relationships
30  new_context = Context(subject_list)
31
32  if old_context is None:
33    # nothing to merge
34    return new_context
35
36  # merge contexts
37  else:
38    for each new_subj in new_context:
39      for each old_subj in old_context:
40              # find a connecting predicate between the 2 subjects
41        connecting_predicate = get_connecting_predicate(new_subj, old_subj)
42
43        # a connection between the subjects exists
44        if connecting_predicate is not None:
45          new_subj.add_relationship(connecting_predicate, old_subj)
```

Listing 5.28: Pseudo-code for context-recalling system

An evident limitation of such an approach is determined by the fact that resources can be eliminated from the context, just in case they are not being associated together with a resource of the new (sentence) context.
If the user were to specify *now the director of the movie, but ignoring its actors*, the tool would try to gather information about the director and their actors, because it does not understand negations.
However, this is also a limitation of the extractor itself, which does not pass the information to the context manager.

5.4.8 Practical problems

The problems that we are confronted to in practice are numerous. Some of them are discussed in section "Well-known Problems" 1.4).
The remaining addressed difficulty, on which the chapter focuses, is the quantity and the quality of information provided by the user. More precisely, our concept of information is identified by the 4 fundamental elements, discussed in subsection 5.4.5.

Missing information Often, sentences do not explicitly contain all the information necessary to identify and connect subjects, objects and predicates in order to form a SPARQL query.
For example, every English speaker is probably capable of identifying the intention of *Tell me the movies by Clint Eastwood*. On the other hand, a program may realize that some *movies* have to be retrieved, and that *Clint Eastwood* might mean something useful.
However, since it is not explicitly mentioned that *Clint Eastwood* equals in fact to the junction of *name* and *family name* of a movie director, and that the titles of the movies are the target of the sentence, the program may have difficulties in understanding the

question.
These aspects are important and a reality when working with Rasa and a limited amount of training data. In fact, the amount of intents that can be recognized are very limited, and providing them manually is a very tedious, slow procedure.

Example Consider the input sentence *what about the ones with Will Smith?*, which clearly relates to a previous context. Rasa cannot identify the intent, because it is not part of the data used to train its model and therefore, the confidence with which it returns an intent is extremely low or equals 0. However, the tool is able to extract some entities: it mistakenly classifies *Will* as the name of a movie, but does a good job by determining that *Smith* is a family name. In this situation, if we were to build a query based on this information, the result set would be empty, as the first name *Will* was matched with the wrong predicate. Listing 5.29 shows the resulting wrong graph pattern, which would be contained in a SPARQL query.

```
1  {
2  ?Film1 rdf:type dbp-owl:Film.
3  ?Film1 dbpprop:name "Will".
4  ?Actor1 rdf:type dbp-owl:Actor.
5  ?Actor1 foaf:familyName "Smith".
6  }
```

Listing 5.29: Faulty graph pattern created by using Rasa

This case-scenario is not acceptable and a workaround needs to be elaborated.

The information is hidden in the sentence Missing information could be found in implicit meanings. For instance, the sentence *tell me the who starred in Gran Torino* implicitly significates *tell me the names of the actors who starred in the film named Gran Torino*. By using the former phrasing, a computer may not return the right data since the predicate *starring* of a movie refers to a resource and not to the names of the actors. The result set would then contain only resource IRIs, which are not meaningful to a human, because the target has not been specified.

5.4.9 Dealing with implicit information

All the methods explained in this section, aim at solving issues related with the problem of missing (implicit) information. Contrarily to other existing solutions, the approaches proposed in this chapter make use of different techniques to relate multiple words within sentences and within contexts. The previously migrated semantic dataset is used to discover the meaning of utterances in the specific topic of film productions and the relationships between other words. More specifically, we try to understand meanings of words by considering their relations inside the RDF graph, similarly to what we did with merging the context.

During the development of the proposed solutions, it was observed that predicates and/or subject types were typically missing.
This is demonstrated with the help of the following example:
Tell me the movies by Clint Eastwood.
The subject's type is missing, as well as the predicates for the objects *Clint* and *Eastwood*.

Intuitively, a human-being would be able to identify that the ideal subject type is *MovieDirector*, and that the proper unspecified predicates are *foaf:name* and `foaf:familyName` for the words *Clint* and *Eastwood* respectively.

Note: At first sight, the predicate `dbpprop:director` might appear to be missing. However the *by* connector would be its equivalent. Although very generic, *by* tends to refer to the main creator of a thing (in this case *MovieDirector*), and not to its participant (*Actor*).

In this chapter, the various issues regarding the four fundamental elements of a query are divided in categories and discussed singularly. However, these factors appear often together. Therefore, the proposed solutions must be joined or backed up reciprocally in order to guarantee better results.

5.4.10 Missing predicate (subjects, objects and target remain)

The user specifies in his sentences three out of four fundamental elements (see 5.4.5):

1. All involved subject types, e.g. movie, actor
2. Some or no filter predicates (missing)
3. Object literal(s) for each of the filter predicates
4. The target, in form of a predicate with no object value specification

A sentence matching this setup could look as follows: *Tell me the name of the director of the movie Titanic*

Note: in this case, the predicate referring to the title of the movie `dbpprop:name` is not explicitly specified, but the literal to be associated with is: "Titanic", which implicitly refers to the title predicate of the movie.

Interpretation of the four fundamental elements from the list above:

1. Subject types: *director*, *movie*
2. Filter predicates: (missing)
3. Object literals: *Titanic*
4. Target predicate: `dbpprop:name` (corresponding to the title of the movie)

A human being aware of the fact that *Titanic* is the title of a film, would certainly be able to guess the missing predicate. In the following sections, 5 possible approaches to try identify the correct missing predicate(s) are explained. The first 4, named with the prefix "A", all make use of some graph metadata while the last one, named with the prefix "B", does not need metadata information.

Solution A1: Graph metadata, choosing a predicate from a candidate list

Graph metadata is information about which subject type (called domain) possesses which predicates. Further, the predicates point to a specific set of object types (called range). This is the equivalent of an ontology, which defines how resources are connected to each

other (see appendix B). This solution aims to solve the problem in which Rasa extracts the wrong entities out of the sentence.

Idea The name of each entity extracted by Rasa is ignored. Contrarily, the key of the solution lies in the *type* of its literal value. For each literal extracted, we look for matches through all possible predicates of the old context. A match is given when the predicate's range of value equals the type of the literal. Such matches are referred to as *candidate predicates*.
For example, the range of the `dbpprop:budget` predicate are integer values. Therefore, the literal "Brad", being a string, does not match. Contrarily, *budget* and the constant integer *53'000* match, therefore this predicate is considered a valid candidate for the literal.
The most suitable predicate is then chosen randomly from the set of candidate predicates.

Our implementation The file `graph_metadata.py` contains hard-coded information about the relations between resources and literals. We believe that a dynamic creation of this data achieved by querying a RDF source is feasible and would allow to add new types of data in the graph (without compromising our solution) effortlessly.

Results In very few situations, where the number of resources in the old context is reduced, the algorithm is able to guess the right predicate.

Limitations Taking advantage of the graph metadata alone is not enough to solve this problem efficiently, due to the high amount of candidate predicates that would match a certain literal type. In fact, considering only the *Film* and *MovieDirector* subject types, there are already 17 predicates connecting them to strings, which are the most common type of literal found in texts. Therefore, the probability of guessing the correct predicate, based only on the metadata, is extremely low.
Moreover, choosing the wrong predicate might return results unrelated with the user's filters or, in most of the cases, no results at all.

Solution A2: Graph metadata, accepting all predicate candidates

Idea The underlying idea is the same as the one adopted for solution A1: given the graph's metadata, collect a list of *candidate predicates*. However, instead of choosing one candidate, the algorithm accepts all candidate predicates as valid, by taking advantage of SPARQL's ability to query for multiple alternative paths simultaneously. This can be realised in the SPARQL query, by using an `OR` expression (denoted by a vertical pipe) between all candidate predicates, as shown in listing 5.30. With this technique, all possibilities are tried and aggregated together automatically in a single result set.

Example Considering the following question:
Tell me the name of the movies starring actor Clint Eastwood. Assuming that the candidate predicates for both "Clint" and "Eastwood" are `foaf:name`, `foaf:familyName` and `dbpprop:profession`, listing 5.30 shows the query resulting by the application of this solution.

The amount of different combinations that the triplestore has to evaluate are 9, due to 3 alternative predicate for both literals.

```
1  SELECT ?name WHERE {
2        ?film rdf:type dbp-owl:Film.
3        ?film dbpprop:name ?name.
4        ?film dbpprop:starring ?actor.
5        ?actor rdf:type dbw-owl:Actor.
6        ?actor foaf:name | foaf:familyName | dbpprop:profession "Clint".
7        ?actor foaf:name | foaf:familyName | dbpprop:profession "Eastwood"
   .
8  }
```

Listing 5.30: SPARQL query with alternative property paths

The main advantage of this approach is that it guarantees that the correct predicate will be part of the query and therefore, of the final solution. Thus, the desired results will *always* be contained in the solution set.
Note: This approach is the equivalent of a mathematical union operation of all the solution sets resulting from applying each predicate singularly.

Our implementation This solution has only been manually experimented through SPARQL queries, without providing a concrete implementation. However, GraphDB was able to react to example queries quickly, even though it had to try multiple path combinations, which consequently increases the cardinality of the result set.

Limitations Since every combination of the predicates is tried, the aggregated result set could contain lots of unwanted data.
For example, if the user was looking for movies starring Michael Jordan, the result set would also include, besides his movies, all the movies where a Jordan Michael (who actually exists) was playing a role.
On the other side, if the solution set was supposed to be empty (i.e. Michael Jordan never played a role in a movie), only the movies of Jordan Michael would be returned instead.
This issue is less concerning when other combinations of predicates are applied. For example, it is much rarer for movie titles to coincide with first names and therefore, unwanted data does not get past this filter.

The gravity of this limitations need to be further investigated, and probably depend on the information domain.

Solution A3: Graph metadata and ask the user for clarification

Idea The underlying idea is the same as the one adopted for solution A2, in which no candidate predicate is discarded (as in A1). In case of insecurity, in which multiple candidate predicates are considered, the system asks the user to confirm a specific subject-predicate-object combination. For example, by asking "Sorry I'm not sure I understand: did you mean to say that a movie has a name Gran Torino ?".
Afterwards, the system might try to learn from the feedback of the user. An applied algorithm for this approach is out of the scope of this thesis and has not been considered. However, there is an existing solution based on the concept of triple clarification named "Querix". According to the paper "Querix: A Natural Language Interface to Query

Ontologies Based on Clarification Dialogs"[22], it is possible to ask the user to clarify the kind of triple that was meant in the sentence, by providing him with a list of options from which to pick.

Example

- "biggest city" means the "city with largest value for `dbpprop:population`"
- "biggest city" means the "city with the largest value for `dbpprop:area`"

The user would then choose the desired meaning of his sentence and receive the most appropriate result based on his decision.

Our implementation No attempt to implement Querix nor a custom solution in the system was made. However, it might be an interesting alternative to experiment, as the user would certainly be able to define what meaning was intended. Hence, a deeper analysis of the tool required.

Limitations The choice of the predicate is not automated and the user will likely have to specify the connection every time he poses a question.
To avoid making this become a tedious task for the user, a confidence rating system would have to be implemented, in order to need user's support just in few, special situations.

Solution A4: Graph metadata and ASK Queries

Idea "Applications can use the ASK form to test whether or not a query pattern has a solution. No information is returned about the possible query solutions, just whether or not a solution exists."[23]
A query pattern is a block of triple statements found in the WHERE clause of a SPARQL query.
The *ASK* query form returns `True` if the query pattern matches something, `False` otherwise.

Hence, no target needs to be specified in the query.

Due to the more aggressive optimization of the ASK queries from the side of the triplestore[24], the ASK form performs faster than the normal SELECT form.

The objective is to discover which subject-predicate-object statements yield a non-empty set, i.e. they exist in the RDF graph. This is done by operating ASK queries on single triple statements for each predicate of the resources from the old context.
If the ASK request returns True, the predicate is kept and considered a candidate between the resource (subject) and the specified literal (object). With this approach, a list of candidate predicates that are known to return a result is constructed. Afterwards, it is possible to proceed in 2 different manners:

1. A random predicate is selected from the list of candidates
2. Additional user clarification can be requested, as explained in Solution A3.

The advantage is that just results for one predicate are returned, contrarily to solution A2, where all the possible predicates where accepted. In most cases, we expect the result set not to be empty.
Further, we expect the technique described to have a positive impact on database performances and network, by not having to bind variables for paths which do not exist.
However, this is outside the scope of the thesis, and further study is required in order to prove this theory.

Our implementation We did not implement this idea, as solution B1 offered a much more pragmatic alternative to this approach, without having to involve the use of metadata.

Limitations Sometimes, an empty solution set is the correct solution. In case the RDF graph contains multiple predicates pointing to the same values, looking for the predicate which necessarily returns a result could reduce the pertinence of the answer, because other predicates can be matched too. However, this depends only on the domain in which this approach is adopted, and necessitates a proper case-study before being implemented.

Solution B: Wildcard predicates and ASK queries (no metadata)

This solution is more similar to the reasoning of a human in comparison to the ones of category "A1" to "A3", because it takes into consideration the connections between real values in the graph.

Idea Instead of a specifying a predicate for a triple statement, SPARQL 1.1 allows for an unknown, arbitrary one to be inserted between a subject and an object, in form of a query variable. In this thesis, this kind of predicate is referred to as "wildcard predicate", or simply "wildcard".
The idea proposed consists in making use of such wildcards in order to avoid having to scan for candidate predicates in the back-end, thus reducing its complexity For every value of the entities returned by Rasa, verify whether a path between it and each resource of the old context exists. If it does, a wildcard relationship between the resource and the literal is added, and no more connections with other resources are tested. After this procedure has been executed for all literals (entity values) extracted by Rasa, the SPARQL query is generated and the result set returned.

Our implementation Our implementation coincides with the procedure explained in paragraph *Idea*. For every literal returned by Rasa, the presence of a connection with a resource of the old context is tested. If the query is positive, it means that a non-empty result set exists, and the relation is added to the new context.

Results Our implementation was able to answer the same questions as those answered by the solutions which were leveraging the graph's metadata. Results in terms of performances could not be estimated.

Limitations This approach increases the complexity of the query to be evaluated from the triplestore, thus decreasing performances, due to the fact that all predicates of a subject are tried, even if their range does not match the literal's type.
The magnitude of the impact on the performances needs to be explored more in detail, before it is possible to make any assessment.

5.4.11 Missing subject (predicates, objects and target remain)

Key point: Since subjects are mapped as IRIs in the graph, the user is never able to specify an exact resource, but is usually able to specify directly connected literal objects which can identify them.
For instance, the combination of name and family name tend to be almost unique identifiers of an Actor, at least from the user's perspective.

Example of missing subject In the sentence *Tell me the movies of Clint Eastwood*, the combination of the name *Clint* and family name *Eastwood* is perceived as unique by the user, even though he is aware that multiple people could possess the same characteristics.

In our eyes, this problem appears to be slightly more complicated in comparison to missing predicates. This is due to the fact that one approach could be contrasting with the solutions adopted for the issue of missing predicates. The method in solution A1 or A2, for instance, consists in cycling through all given subjects of the context and trying to match to every literal type.
Additionally, other solutions that could be applied for the problem discussed, would interfere with the proposed solutions of missing predicates.

Solution C: Arbitrary subjects and predicates

Contrarily to the previous proposed solutions, letter C introduces the concept of avoiding Rasa completely, since it is burdensome to handle faulty entities from the tool. The approach tackles an additional problem with Rasa that was not discussed previously. Not only can the entities be matched with the wrong predicate, but their literals could also be totally missed by the tool, as they are often considered cluttering utterances in the input sentence. Therefore, an approach for backing up and even substituting Rasa's entity extraction functionality is explained in this section.

Idea Taking inspiration from solution B of the problem regarding the missing predicates, a solution has been proposed.
The basic idea is similar to the approach to solve mathematical problems, that is, to ignore real information as much as possible. Therefore, we will be focusing more on the structure and dependencies of unknown variables, rather than on their true values. This is done by:

1. POS-Tagging the words in the sentence (e.g. with NLTK)
2. Removing cluttering utterances, such as stop-words and connectors, by accepting only the following tags: CD (cardinal number), JJ (adjective), JJR (adjective, comparative), JJS (adjective, superlative), NN (noun, singular), NNP (proper noun, singular), NNPS(proper noun, plural) and NNS (noun, plural).

3. With the help of SPARQL, specifying arbitrary connections between arbitrary subjects and the remaining words (such as “Tom” or “Horror”)
4. Defining a graph pattern of arbitrary connections of an arbitrary length (0 to infinite), which may or may not exist, between all the arbitrary subjects

An example of the resulting graph pattern can be represented in form of verbal statements:

- an arbitrary resource ?x is connected to “Tom” through an arbitrary predicate ?px1
- ?x is also *possibly*, somehow (regardless how) connected to an arbitrary resource ?y
- an arbitrary resource ?y is connected to “Cruise” through an arbitrary predicate ?py1
- ?y is also *possibly*, somehow (regardless how) connected to an arbitrary ?z
- ?z ...

Listing 5.34, discussed in the following example, represents this idea in form of a graph pattern.

When the confidence is too low (i.e. under a certain defined threshold), the entities extracted by Rasa are ignored and the full-text is considered. Contrarily, when the confidence is over the aforementioned threshold, we assume that the entity extracted by the tool might be accurate enough. Consequently, only the chunk of text excluding the extracted entities will be used.
The first true step of the idea consists in POS-tagging the (remaining) items in the sentence.
The result of this operation for the case in which the confidence is too low is demonstrated in listing 5.31. Alternatively, supposing that the confidence might be high enough, the state occurring after POS-tagging only the non-extracted items is shown in listing 5.32. *Note:* this is just for instruction, it does not represent the real case scenario, and will not be explored further in this example. In fact, the confidence that Rasa returns is indeed too low.

```
1 (what, WP), (about, IN), (the, DT), (ones, NNS), (with, IN), (Will, NNP),
    (Smith, NNP)
```

Listing 5.31: POS-tagging on the full sentence; because confidence is too low to trust the extracted entities

```
1 (what, WP), (about, IN), (the, DT), (ones, NNS), (with, IN)
```

Listing 5.32: POS-tagging on the items remaining in the sentence

The second step consists in removing cluttering words. Every word whose POS-Tag does not equal one of the accepted tags is removed. The remaining words would represent a sequence of possibly useful keywords, each of which can potentially be matched to a subject type, a predicate (belonging to a specific subject type) or a literal object (either a number or a string contained in the RDF graph).
The result of this operation is shown in listing 5.33. It can be seen that *Will* and *Smith* are part of the list, but *ones* does not help our cause. However, with this approach this is not a problem, because a path to *ones* is allowed to either exist or not.

```
1 (ones, NNS), (Will, NNP), (Smith, NNP)
```

Listing 5.33: Removed cluttering words based on their POS-tag

The third step is connecting arbitrary subject to the remaining literals. The resulting graph pattern is displayed in listing 5.34.

```
1   ?x ?px1 "ones".
2   ?y ?py1 "Will".
3   ?z ?pz1 "Smith".
```

Listing 5.34: Graph pattern resulting from connecting arbitrary subjects to their literals

We have just stated that there are three, possibly (but not necessarily) distinct resources which possess a single, arbitrary connection of length 1 to the respective literals.

The fourth and final step (excluding query generation), is to connect the various subjects with paths of arbitrary length. This is usually done with the asterisk (*) symbol, which means "0 or more repetitions of the specified predicate". However, SPARQL does not allow arbitrary lengths when the predicate is a variable, such as `?px1`. Therefore, a trick is used, whose syntax does not need to be understood: `!(<>)`. This simply allows for any predicate to be fitted at its place, and it is possible to add a concluding asterisk, in order to consider basically every possible path from the subject to the object: `!(<>)*`.
The final result of this operation is shown in listing 5.35.

```
1   ?x ?px1 "ones".
2   ?y ?py1 "Will".
3   ?z ?pz1 "Smith".
4
5   #newly added, arbitrary connections between resources
6   ?x !(<>)* ?y.
7   ?y !(<>)* ?z.
```

Listing 5.35: Graph pattern resulting from connecting arbitrary subjects to their literals

At this point, the context corresponding to this graph pattern can be linked with the old one.

Implementation This solution has not been implemented, due to conflicts regarding the choice of the predicate with the adopted solution B. Additionally, the complexity of the query was too high, as explained in chapter the next paragraph.

Limitations Complexity: the asterisk (*) allows for paths to be arbitrarily long. Combined with the fact that `!(<>)` matches any predicate, the asterisk causes the pattern to match *every* possible combinations of connections between the subject and the specified object.
Moreover, the subject can be of *any* type, that is, every resource contained in the graph could represent this subject.
Both Jena and GraphDB were not able to solve such a query.

Outlook The amount of unknown information and possible combinations is too high to compute with the asterisk symbol. In this paragraph, a few ideas for the reader are proposed. These might lead to a more efficient use of the approach explained in this section.

Use question mark (0 or 1) instead of asterisk (0 or more) Information that relates together in sentences is usually also tightly coupled in the domain modeled. Therefore, it might be possible to make use of the question mark symbol (?) instead of the asterisk (*), allowing the arbitrary predicate (trick: `!(<>)`) to appear "0 or 1 times".

"Divide and conquer" Possibly, information regarding each subject could be gained separately, which allows to create less vague queries. The various sub-patterns deriving from this procedure could easily be joined together, thanks to the characteristics of SPARQL.

5.4.12 Missing target information (subjects, objects and predicates remain)

The user specifies in his sentences three out of the four fundamental elements for a successful query generation:

1. All involved subject types, e.g. movie, actor
2. All filter predicates
3. Object literal(s) for each of the filter predicates
4. No target predicate

An example question for this case is *now tell me its actors*, which clearly refers to a previously created context. Supposing that "its actors" gets translated into a "starring" predicate, the user would receive an IRI representing the actor resource.

Problem

The main issue with a missing target predicate is presented when resources are selected as the target. In comparison to literals, resources are identified by an IRI instead of their value, such as a string or a number. In this case, although the resource IRI could be returned to the user, it would be meaningless in the eyes of a human-being. For him, a person is often identified by their name and lastname, and not by an URL.
Therefore, the objective in this chapter is to find always return meaningful results to the user.

Solution D: Fixing the R2RML mappings

The issue of the IRI not possessing a meaningful identifying value is exacerbated by the bad modeling approach of our RDF database, due to our lack of experience in the field: a standard practice to point to identifying literals through the `rdfs:label` predicate[25]. With such a design, specifying the identifying predicates for each resource would not be

necessary, as that information would be stored in the `rdfs:label` property. The information could then be retrieved automatically every time that the resource is requested. Fixing the R2RML mapping files would solve this problem efficiently, as `rdfs:label` is "[...] used to provide a human-readable version of a resource name"[25].

Implementation This solution has not been implemented, as it would imply the regeneration of a the raw triples.

Limitations The identifying values have to be defined at the moment in which the mapping file is parsed.

Solution E: Default values for resources

A possible solution is to define some default, identifying values for each resource that will be selected in case of a request containing an implicit target. A *target* refers to the value that the user would like to know.
For example, an identifying value for a movie would be the title (`dbpprop:name`), while an identifying value for a person could be the combination of its name and lastname (or a special nickname which they are often identified with, such as "The Rock").

Implementation As a quick patch, we implemented this solution directly in our hard-coded metadata. We then inserted a small check that is run when the context is about to get transformed into query, that adds the default values as targets in case no other target was specified for the resource.

Limitations The default predicates have to be set in the hard-coded metadata. That means, that when new resources are inserted into the domain, the metadata would need to be updated.

Solution F: Have the IRI pointing to a REST endpoint

Alternatively, having the IRI pointing at a REST accessible resources would also solve the problem, when combined with a proper UI handling, in which the predicates and the respective constant values are displayed. An advantage of this system is the possibility for the user to browse for more than the `rdfs:label` values, giving him the freedom of choice.

Implementation This solution has not been implemented, as it would have been too time-consuming.

Limitations Requires the user interface to be aware of the technique used and to be able to represent information regarding every type of resource properly.

Chapter 6

Results

General infrastructure
We have observed relevant architectural components concerning chatbots. The modularity proposed in this thesis has enabled us to introduce and test new components without many difficulties. There were no major problems when we swapped Quepy with Rasa NLU. We are satisfied with the front-end. When taking into consideration the time invested in re-using the premade template and implementing the few example functionalities (youtube video and charts), we can be more than happy with the results.

Data
Data plays a very important role in information retrieval solutions. We have seen that the number of dataset about movies is limited and they are not ready-to-use. Instead, they require a reasonable amount of work to be usable. Based on the experience with Quepy, we realized that building a tool which dynamically writes SQL queries is challenging. These are not made to be simple but to have much more control as possible, allowing very complex queries to be written. On the other hand, SPARQL queries have proven to be simpler even for a computer program, as they consist of minimalistic statements. Instead, using a system which is easier to query makes much more sense since it decreases the complexity of the queries. We have demonstrated that even with a limited amount of time, it was possible to build a tool which is able to write valid SPARQL queries.

Quepy
Quepy's limitations have been assessed, the most serious one being the tendency of question templates to overlap, which is proportional to the amount of question templates and the number of different formulations that each template is able to match. We were able to provide a first concept for extending the tool with FILTERs and ORDER BY modifiers.

IMDb raw files to MySQL
IMDBpy is a viable tool for converting the IMDb raw (text) data into a MySQL database, which summed up for 15 GB of data. The conversion alone required 322 minutes (5 hours and 22 minutes), out of which 40 were necessary for writing the data, 264 for creating the foreign keys and 17 for the indexes.
This operation was executed on a laptop with an Intel Quad-Core i7-4710MQ processor, 8GB RAM and a SSD.
To avoid having to repeat the conversion with IMDBpy on each device, a dump of the

MySQL IMDb database was made. Importing the dump file on another computer, however, was an equally slow operation. In fact, on a MacBook Pro with an Intel Core i7 processor, 8GB RAM and a SSD, importing the dump required approximately 5 hours.

From relational to Turtle triples
Transforming a big relational database into a RDF graph is time consuming, as the market of RDB2RDF solutions is not yet established and the performances of existing ones lack of optimization.
The R2RML standard and R2RML Parser tool are viable options for such conversion, although some workarounds need to be adopted, especially related to the limitations of the tool and not of the standard itself.
The machine on which the conversion takes place needs to have sufficient RAM capacity. Besides, the splitting of the conversion process into multiple R2RML mapping templates is still necessary.
The amount of RAM needed by the conversion tool depends on the amount of triples produced in a single mapping file, rather than on the size of the whole, normalized database. 1 GB of RAM is needed every approximately 107'000 triples to be generated. The cluster we had at disposition required a total of approximately 8 hours to generate around 35 million triples from all the mapping files using R2RML Parser.

Triplestore performances
GraphDB has faster import and query times in comparison to Jena. The latter, was unable to perform very complicated queries, where the amount of arbitrarily valid predicates was elevated. The causes for this difference in performances are out of the scope of this thesis and need to be researched further.
While Jena required approximately 6 hours to import the raw triples through command line (tdbloader2), GraphDB was able to execute the same operation in 7 minutes from the web interface.
Further, GraphDB proved to be able to answer an extremely complex query in 5 seconds, while Jena was not able to return an answer (operation has been interrupted after 3 minutes).

Rasa NLU The amount of intent training data at our disposition was non-existent, and we had write examples on our own. For the sake of this thesis, we did not concentrate our efforts into producing a great amount of intents to train Rasa. Therefore, to optimize this system, further collection or production of training data is required to allow a proper evaluation of the system.
Nevertheless, Rasa NLU has shown to be a valid and flexible tool.

Context-recalling Effective information retrieval with regard of the context in human-computer conversation is still at its blooming phase. Due to the severe time constraints combined with the vastness of the covered topics, no scientifically provable results could be provided. Additionally, evaluation of the effectiveness of the implemented solution cannot be provided without a standardized question-answer dataset for comparison, which is to date still missing.
The lack of a proper comparative system is due also to the domain in which the solution takes place, as its performances depends heavily on the branch in which it is applied (see [5]).

Chapter 7

Discussion and conclusions

The objective for this thesis was set to be a prototypic chatbot, paying attention to the entire architecture.

In this thesis, many tasks, techniques and solutions have been either proposed or implemented. However, due to the primitive state of the chatbot and the lack of a comparative dataset on which to perform comparisons and evaluations, this chatbot cannot be yet considered practicable, although some interesting solutions have been proposed.

Without an appropriate comparative dataset, a scientific evaluation of the quality of the chatbot cannot be made. Therefore, the most important task ahead, is certainly the development of a comparative set of questions and answers for the domain of film productions. The wider and the more precise is the definition of the requirements and the scope in which the test sets operate, the more recognition from the public the chatbot can gain. It is in fact clear, that even the best algorithm in the world needs to be evaluated and proved in a structured fashion, before being considered such.

From the very beginning, the cardinal objective was to implement the complete infrastructure of a chatbot. This way, the prototype could have constituted the basis for future developments in this field. However, we regret not having saved the time invested to experiment various technologies for more direct and relevant purposes, such as the research or creation of a comparative dataset.
Learning new tools and technologies can be time expending, and when dealing with many of them, the amount of effort required to master them and learn their strengths and weaknesses is huge. Nevertheless, the process was undoubtedly necessary in order to be able to extend the tool used.

Lack of experience contributed negatively to our choices throughout the whole duration of the project. However, for the future, we set ourselves to be more pragmatic about our decision and to maintain a clear vision of the objectives.

Outlook

There are many extensions and modifications that can be applied on the components of our prototype. Surely, one of the most urgent tasks is the addition of support for new intents and new training examples for the existing ones. This would improve the range of questions that a user can ask, thus reducing the disappointment for receiving a negative answer, as this would happen more rarely.

Our techniques to identify elements which will compose a triple statement inside of a text displayed many limitations in doing so. Thus, this is an area in which many improvements can be made, as proved by the huge amount of possible solutions that have not been implemented, as well as some ideas that are popping out on the Internet.

One of the techniques that we discovered towards the end of the thesis, but that we did not have time to explore more in detail, is called "Semantic Role Labelling" (SRL). "The primary task of semantic role labeling (SRL) is to indicate exactly what semantic relations hold among a predicate and its associated participants and properties [...]" [26]. We believe that this technique can be used to identify triples in a sentence, which could then be converted into a SPARQL query able to retrieve information from a RDF graph. This technique needs to be researched and eventually implemented, in order to be able to make a statement regarding its feasibility in practice.

Appendix A

Project management

Meetings

Exceptions apart, weekly meetings took place with our supervisor Dr. Mark Cieliebak. It was particularly useful to note down in an aggregated document all key aspects of the discussion. Further, arriving prepared at the meeting always allowed us to be more effective and clear about the objectives that have been achieved, the problems encountered and their corresponding solutions that were adopted. Each time we did not arrive well prepared, it was burdensome to find the right expressions and explain our ideas to the supervisor. We regret these episodes, and we believe that it also influenced us negatively and brought us on the wrong road.

Each meeting concluded with an outlook on the possible directions in which to continue developing. This was particularly useful for being able to set up mini-objectives for the following week. However, it has been sometimes challenging to keep an eye on the global vision, and we ended up falling out of the desired track.

Timetable

On the following page, figure A.1 displays the effective timetable. The numbers refer to the workload, on a scale from 1 to 6.

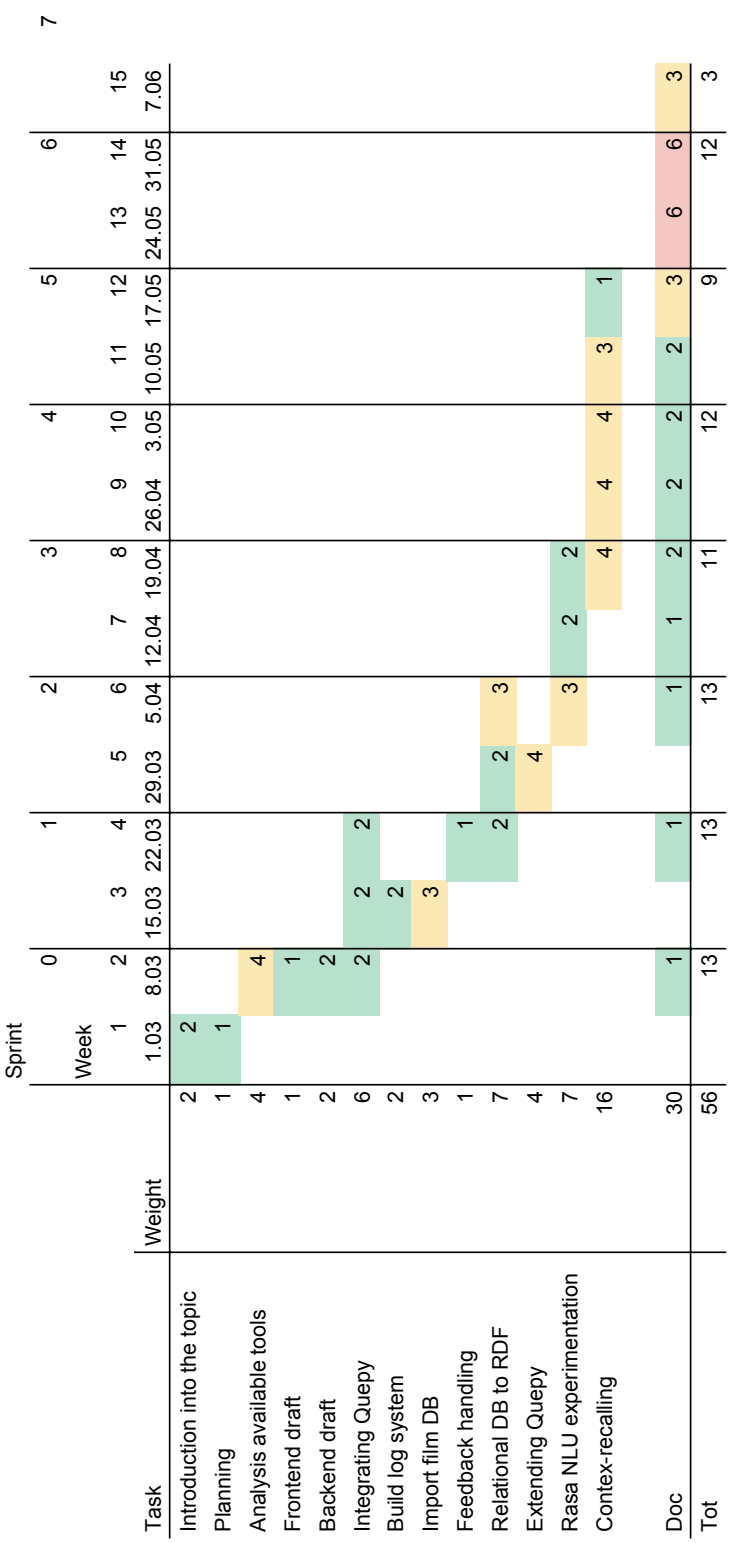


FIGURE A.1: Effective timetable

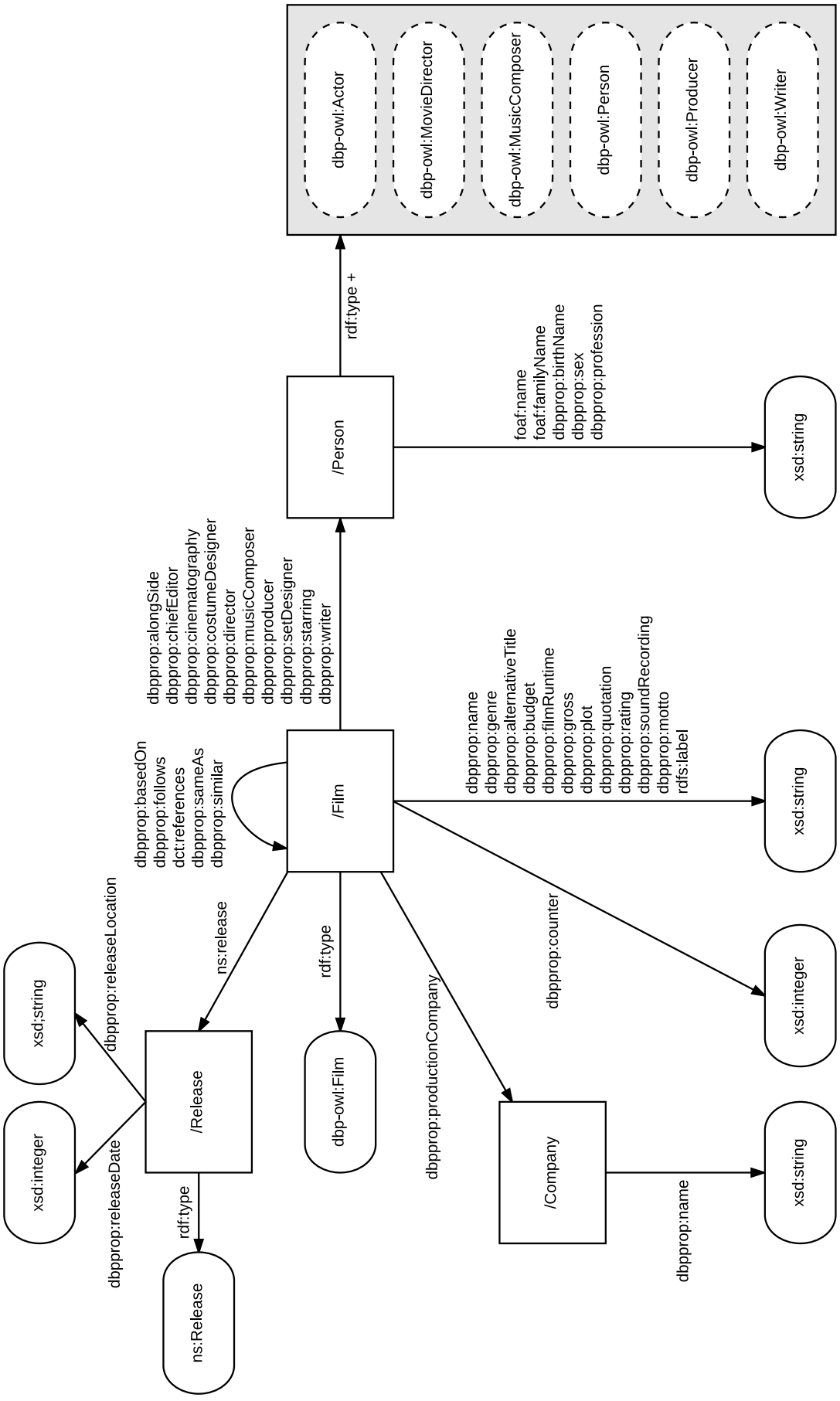
Appendix B

IMDB RDF Database schema

The representation in the following page is not subject to any standard.

Interpreting the RDF Schema

- Rectangular shapes with inner texts starting with a forward slash represent resources, i.e. elements identified by an IRI. They can represent either subjects or objects of a triple relationship.
- Round shapes with a solid border contain the type of value that the predicates are pointing to.
- Round shapes with a dashed border contain the IRI value they can assume.
- Round shapes in general are able to assume only the role of the object in a triple statement.
- The arrows represent the predicate between subjects and objects in a triple.
- The objects stay on the side of the arrow.
- Every new line of text in the edges represents an alternative predicate for that type of connection.
- A plus symbol (+) next to a predicate implies the condition that at least one statement containing such predicate exists for the given subject to any of the given objects.
- A grey rectangular box around object IRIs specifies an 'OR' condition between all the contained objects.


dbp-owl:Actor
dbp-owl:MovieDirector
dbp-owl:MusicComposer
dbp-owl:Person
dbp-owl:Producer
dbp-owl:Writer
rdf:type +
/Person
foaf:name
foaf:familyName
dbpprop:birthName
dbpprop:sex
dbpprop:profession
xsd:string
dbpprop:alongSide
dbpprop:chiefEditor
dbpprop:cinematography
dbpprop:costumeDesigner
dbpprop:director
dbpprop:musicComposer
dbpprop:producer
dbpprop:setDesigner
dbpprop:starring
dbpprop:writer
dbpprop:basedOn
dbpprop:follows
dct:references
dbpprop:sameAs
dbpprop:similar
/Film
dbpprop:name
dbpprop:genre
dbpprop:alternativeTitle
dbpprop:budget
dbpprop:filmRuntime
dbpprop:gross
dbpprop:plot
dbpprop:quotation
dbpprop:rating
dbpprop:soundRecording
dbpprop:motto
rdfs:label
xsd:string
ns:release
rdf:type
dbp-owl:Film
dbpprop:counter
xsd:integer
dbpprop:productionCompany
/Company
dbpprop:name
xsd:string
dbpprop:releaseLocation
xsd:string
dbpprop:releaseDate
xsd:integer
/Release
rdf:type
ns:Release

Appendix C

Part-of-speech tags

Note: Only the most important tags have been listed

Tag	Description	Example
CD	Cardinal number	mid-1890 nine-thirty forty-two ten million 0.5 1987 two 78-degrees IX ...
DT	Determiner	all an another any both each either half much some such that these this
IN	Preposition or subordinating conjunction	astride among whether out inside despite on by throughout ...
JJ	Adjective	third ill-mannered pre-war regrettable oiled calamitous first separable multi-lingual ...
JJR	Adjective, comparative	briefer brighter broader calmer cheaper ...
JJS	Adjective, superlative	calmest cheapest choicest classiest deepest ...
NN	Noun, singular or mass	cabbage knuckle-duster Casino afghan thermostat investment hyena override machinist ...
NNP	Proper noun, singular	George, Angelina, Eastwood, Mike, Michael, Apple, Motown Oceanside Liverpool Kreisler ...
NNPS	Proper noun, plural	Amusements Anarcho-Syndicalists Andalusians Andes Andruses Antilles Antiques ...
NNS	Noun, plural	undergraduates products bodyguards facets coasts clubs fragrances averages
RB	Adverb	occasionally maddeningly professedly prominently ...
TO	*to*	to
VB	Verb, base form	assemble assess assignbank begin believe bend benefit bomb boost brush build ...
VBD	Verb, past tense	dipped swiped regummed soaked tidied exacted aimed adopted contemplated ..
VBG	Verb, gerund or present participle	telegraphing stirring focusing angering judging stalling approaching traveling ...
VBN	Verb, past participle	multihulled dilapidated aerosolized chaired languished panelized used experimented dubbed desired ...

TABLE C.1: Part-Of-Speech tags and corresponding example words (cf. [3], [4])

Appendix D

Installation

In order to install the complete system, you need to install and configure Rasa NLU, MongoDB and GraphDB. *virtualenv* and *Docker* are required.

D.1 Rasa NLU in a virtual environment

Install Anaconda: https://www.continuum.io/downloads

Prepare the environment
Update anaconda:
```
conda -V
conda update conda
```

Get Rasa NLU:
```
git clone https://github.com/RasaHQ/rasa_nlu.git
cd rasa_nlu
```

Create a virtual environment with Anaconda:
```
conda create -n venv python=2.7 anaconda
source activate venv
```

Install Rasa NLU (it takes several minutes):
```
python setup.py install
```

Install MITIE:
```
pip install git+https://github.com/mit-nlp/MITIE.git
```

Train the model
Download the MITIE models[1] and extract the file `total_word_feature_extractor.dat` in your computer.

Create the file `config.json` and configure the `mitie_file` and `data`. `path` defines the directory in which the trained models will be stored.

[1] https://github.com/mit-nlp/MITIE/releases/download/v0.4/MITIE-models-v0.2.tar.bz2

```
{
  "pipeline": "mitie",
  "mitie_file": "/path/to/total_word_feature_extractor.dat",
  "path" : "./models",
  "data" : "/path/to/moviebot.json"
}
```

Note: If you get any error, try `pip install future --upgrade`

Train the model using:
`python -m rasa_nlu.train -c config.json`
Otherwise you can use our simple model that can be found in `movie_model.zip`.

Run the Rasa NLU server
Start the Rasa NLU server:
`python -m rasa_nlu.server -c config.json --server_model_dirs=./name_of_the_model`
Note: `name_of_the_model` is the name of a folder inside `./models`.

D.2 MongoDB

MongoDB is used for logging and to retrieve back a previous context.

Make sure to change `/path/to/log` to match your requisites and then create a new container:
`docker run --name logger -v /path/to/log:/data/db -p 27017:27017 -d mongo`

Note: There is no need to create a database and a collection since our chatbot does it by itself the first time it is launched.

You can simply start it again using:
`docker start logger`

D.3 GraphDB

Make sure to change `/path/to/graphdb` to the folder which contains the triples and then create a new container:
`docker run -p 7200:7200 --name graphdb`
`-v /path/to/graphdb/:/opt/graphdb/home -t ontotext/graphdb:8.0.4-se`

D.4 Install this project

Note: We suggest to create a virtual environment with Python 2.7

Install the requirements:
`pip install -r requirements.txt`

Copy the configuration file example and edit the necessary entries:
`cp config.yaml.example config.yaml`

Run the server:
`python server.py`

You can now visit http://localhost:5000/ and use the bot.

Appendix E

Contents of the CD

- Copy of this document
- Source code (including R2RML Parser)
- Timetable
- R2RML mappings

Bibliography

[1] "Description of the SQL movie databse made by IMDBpy." https://stackoverflow.com/a/25938338, 2014. [Online; accessed 10-March-2017].

[2] J. Sequeda, "R2RML: RDB to RDF Mapping Language." https://www.slideshare.net/juansequeda/rdb2-rdf-tutorial-iswc2013, 2013. [Online; accessed 09-March-2017].

[3] U. of Pennsylvania, "Alphabetical list of part-of-speech tags used in the Penn Treebank Project." https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html. [Online; accessed 15-March-2017].

[4] E. Atwell, "Automatic Mapping Among Lexico-Grammatical Annotation Models (AMALGAM) ." http://www.comp.leeds.ac.uk/amalgam/tagsets/upenn.html. [Online; accessed 15-March-2017].

[5] K. Kuligowska, "Commercial chatbot: Performance evaluation, usability metrics and quality standards of embodied conversational agents," *Professionals Center for Business Research 02(2015)*, February 5th, 2015.

[6] L. Le Bigot, E. Jamet, and J.-F.-O. Rouet, "Searching information with a natural language dialogue system: a comparison of spoken vs. written modalities," *Applied Ergonomics*, vol. 35, pp. 557–564, 2004.

[7] S. Navarre and H. Steiman, *Natural Language Processing with Python.* O'Reilly Media, Inc, 2002. p. 28.

[8] "Ontotext - What is RDF Triplestore?." https://ontotext.com/knowledgehub/fundamentals/what-is-rdf-triplestore/, n.d. [Online; accessed 10-April-2017].

[9] "DBpedia Ontology." http://wiki.dbpedia.org/services-resources/ontology, n.d. [Online; accessed 10-March-2017].

[10] "OMDB - Monthly Database Dump." http://beforethecode.com/projects/omdb/download.aspx, n.d. [Online; accessed 05-March-2017].

[11] "IMDb, Conditions of Use." http://www.imdb.com/conditions, n.d. [Online; accessed 05-March-2017].

[12] "DBPedia Ontology - Film." http://mappings.dbpedia.org/server/ontology/classes/Film, n.d. [Online; accessed 30-March-2017].

[13] N. Konstantinou, D. Kouis, and N. Mitrou, "Incremental Export of Relational Database Contents into RDF Graphs," *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics (WIMS14) - WIMS '14*, no. June, pp. 1–8, 2014.

[14] R. W. Group, "RDB2RDF." https://www.w3.org/2001/sw/wiki/RDB2RDF, 2012. [Online; accessed 09-March-2017].

[15] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers, "RDF 1.1 Turtle." https://www.w3.org/TR/turtle, 2014. [Online; accessed 10-March-2017].

[16] S. Das, S. Sundara, and R. Cyganiak, "R2RML: RDB to RDF Mapping Language." https://www.w3.org/TR/r2rml, 2012. [Online; accessed 09-March-2017].

[17] "LargeTripleStores - GraphDB^{TM} by Ontotext." https://www.w3.org/wiki/LargeTripleStores, 2016. [Online; accessed 01-April-2017].

[18] D. Moisset, "Querying your database in natural language." https://www.slideshare.net/PyData/querying-your-database-in-natural-language-by-daniel-moisset-pydata-sv-2014, 2014. [Online; accessed 01-March-2017].

[19] R. Carrascosa, "support for sql_92." https://github.com/machinalis/quepy/issues/32#issuecomment-197316096, 2-4 2016. [Online; accessed 05-March-2017].

[20] "Rasa NLU - Processing Pipeline." https://rasa-nlu.readthedocs.io/en/stable/pipeline.html, 2016. [Online; accessed 10-March-2017].

[21] A. K. Dey, "Understanding and using context," *Personal Ubiquitous Comput.*, vol. 5, pp. 4–7, Jan. 2001.

[22] E. Kaufmann, A. Bernstein, and R. Zumstein, "Querix: A Natural Language Interface to Query Ontologies Based on Clarification Dialogs," *In: 5th ISWC*, no. November, pp. 980–981, 2006.

[23] "SPARQL Query Language for RDF." https://www.w3.org/TR/rdf-sparql-query, 2008. [Online; accessed 02-April-2017].

[24] "SPARQL Query Language for RDF." http://blog.dydra.com/2011/05/13/query-tuning, 2011. [Online; accessed 11-May-2017].

[25] D. Brickley and R. Guha, "Resource Description Framework(RDF) Schema Specification." https://www.w3.org/TR/PR-rdf-schema, 1999. [Online; accessed 08-May-2017].

[26] L. Màrquez, X. Carreras, K. C. Litkowski, and S. Stevenson, "Semantic role labeling: An introduction to the special issue," *Computational Linguistics*, vol. 34, no. 2, pp. 145–159, 2008.