A Virtual Assistant for Web Dashboards

Analytics Bot

Anisa Shahidian

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Dr. Angela Bairos Pimentel

Examination Committee

Chairperson: Prof. Ana Maria Severino de Almeida e Paiva
Supervisor: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Member of the Committee: Prof. Ricardo Daniel Santos Faro Marques Ribeiro

October 2018

Acknowledgments

A huge thank you to Professora Luísa. You always remove any lingering of worry that I may have, and I truly could not have chosen a better advisor. I do not have words to describe how helpful you are. Thank you.

Thank you Angela (and José), for answering each and every question about the thesis environment, and always ensuring I had all the tools needed for this project.

I would like to thank my parents, who took care of me all these years, and have provided me with all sorts of experiences. I hope I have made you proud. I would also like to thank you for giving me two sisters. Nika and Sama, you have taught me how to be a friend and an older sister, and I will always be here for you.

Thank you Diogo. For everything. You have helped me grow and given me the encouragement and support to be myself. Life by your side is all and more than I could have hoped for.

I would like to thank Mariana, Chico, Henrique, Pacheco, Zé, Luis, Bruno and Rui. Thank you for all the good times we have had. Having you as my friends has made it all the harder to leave. Also to Mariana, I am glad I said "hi" enough times for you to respond back. Thank you for always being silly with me.

Thank you to Alex and Inês. We have had many memorable late night philosophical talks. I hope your lives will be filled with happiness. Inês, thank you for surviving RRR with me. We've had good times.

Thank you Teresa, Rui, Zita, avó João, tia Lurdes and tio João. You have welcomed me very dearly, and that warms my heart.

Abstract

This thesis relates the research and implementation of a Natural Language Interface to a Database about *Rollover Opportunites*, responsible for answering questions at BNP Paribas Global Client Analytics Group. We study several Natural Language Interfaces to Databases' and other Natural Language Processing systems, taking into consideration the project at hand and the limitations of the problem this thesis deals with. After that, we describe the architecture for the implementation. The environment in which the questions are asked, named entity recognition, the processing of the user's question, and the two approaches implemented are also detailed. We have implemented a semantic grammar, which constitutes sys1. An implementation with the SEMPRE toolkit and its learning component constitutes sys2. Both implementations are detailed, as are the aspects that needed special attention, such as the use of date expressions in the user's question. Our results show that knowledge of the domain is crucial in a rule-based implementation, as it is not flexible. We also notice that our learning implementation, sys2 has a better result than sys1.

Keywords

Information; Natural Language Interface to Database; Natural Language Processing; Interface; Rule Based Grammar; Named Entity Recognition; Database; Table; Query

Resumo

Esta tese relata a pesquisa e implementação de uma Interface de Língua Natural para Base de Dados sobre *Rollover Opportunities*, responsável por responder a perguntas no Banco BNP Paribas, através de pedidos a uma base de dados. Estudamos várias Interfaces de Língua Natural para Base de Dados e outros sistemas de Processamento de Língua Natural, tendo em consideração o projecto e as limitações do problema sobre o qual esta tese se debruça. Seguidamente, iremos descrever a arquitectura para a implementação. O ambiente no qual as questões são inquiridas, o reconhecimento de entidades mencionadas, o processamento das questões do utilizador, e as duas abordagens também são descritas. Implementámos uma gramática semântica, que constitui o sys1. Uma abordagem com a ferramenta SEMPRE e o seu mecanismo de aprendizagem constitui o sys2. Ambas as implementações são detalhadas, tal como os aspectos que necessitaram de atenção especial, como a utilização de expressões temporais na pergunta do utilizador. Os nossos resultados demonstram que conhecimento do domínio é crucial numa implementação baseada em regras, pois a mesma não é flexível. Também observámos que a nossa implementação com aprendizagem, sys2 obtém melhores resultados do que sys1.

Palavras Chave

Informação; Interface de Língua Natural para Bases de Dados; Processamento de Língua Natural; Interface; Gramática baseada em Regras; Reconhecimento de Entidades Mencionadas; Base de Dados; Tabela; Pesquisa

Contents

List of Figures

List of Tables

Listings

Acronyms

NLI	Natural Language Interface
NLP	Natural Language Processing
NL	Natural Language
FOL	First Order Logic
SQL	Structured Query Language
LAT	Lexical Answer Type
NER	Named Entity Recognition
NLIDB	Natural Language Interface to Database
MRR	Mean Reciprocal Rank
GCA	Global Client Analytics
FX	Foreign Exchange
POS	Part of Speech

1

Introduction

Contents

1.1 Motivation

BNP Paribas' Global Client Analytics (GCA) group relies on visual dashboards to interpret the data present in their database sources. Some information can be presented in graphs and charts, however, to obtain more context information about a certain *Client* or *Deal*, one has to look for the corresponding database tables.

This thesis was done in partnership with BNP Paribas and we aimed to simplify this search by creating a NLIDB that allows users to obtain information (present in the database table) in a practical way, without querying a database, or requesting changes on the dashboard.

The dashboard for this project, entitled *Rollover Opportunities*, enables the user of the system (a marketeer or analyst) to look for information pertaining the deals that will expire in the near future.

1.2 Goal and Requirements

The goal of this dissertation is to design, implement, and evaluate a NLIDB for the *Rollover Opportunities* dashboard in the bank *BNP Paribas*. It will accept a Natural Language question, analyze it, search for the answer, and return it to the user. The system can be divided into five components, where different goals are expected. These components are:

- Data Management

  This component is mainly executed outside of runtime, and is responsible for:

  - Obtaining the database table *vocabulary* to reduce ambiguities when creating the query;
  - Creating *synonyms* based on the studies of the database and its usage;
  - Creating *date expressions*.

- Question Analysis

  In Question Analysis, we have the following tasks:

  - Process the question using Natural Language Processing (NLP) methods;
  - Use the *vocabulary*, *synonyms* and *date expressions* implemented in *Data Management* to obtain the possible different interpretations.

- Query Construction

  Query Construction constructs the query[1] that will be submitted to the information sources through

[1] A query is a request, or question. In the case of this project, it will refer to the system's request to the database.

two approaches.

- Answering Step

  This step presents the answer obtained by the query.

- Learning

  The learning mechanism will be implemented in our second approach and takes into consideration training data to chose the correct answer.

1.3 Contributions

The GCA group has been interested in expanding and implementing chatbots. For this, a series of studies and tests needed to be developed so that the best and most practical approach could be found.

This project studies how a NLIDB can be implemented without a large amount of data to gather conclusions from. It focuses on two different rule-based approaches, allowing the GCA group to have an example of what can be accomplished using NLP techniques.

1.4 Document Structure

The remainder of this dissertation is organized as follows:

- Chapter 2 presents work related to this project.
- Chapter 3 details the architecture and the implementation of the NLIDB.
- Chapter 4 evaluates the implementation.
- Chapter 5 concludes the dissertation, and points the main contributions. We also mention future work on this topic.

2

Related Work

Contents

2.1 Introduction

This chapter details the research for this project. We start by describing the historical background of NLIDBs in Section 2.2. We will then look at the main components of NLIDB's in Section 2.3. Section 2.4 will focus on the main component *Data Management*, while Section 2.5 focuses on *Question Analysis*, Section 2.6 on *Query Construction*, Section 2.7 on the *Answering Step*, and Section 2.8 on the *Learning* component. We conclude this chapter in Section 2.9.

2.2 Historical Background

Several systems have tried to provide Natural Language (NL) interpretation over the past decades [1]. Some of them have been classified as NLIDB.

NLIDBs receive a question in NL, and query a database or database table using the meaning of the question. In this work, we will refer to the NL question as a *user question* since they are submitted to the system by the user. In some cases, the *user question* is first transformed into an expression and only then into a query for the database.

The two best known NLIDBs from the 1980's were CHAT-80 and ASK:

- CHAT-80 [2] was implemented in Prolog. It transformed the *user questions* into logical expressions, and then evaluated them against the database. Both the logical expressions and the database were implemented in Prolog, so the logical expression had to be transformed into a Prolog form to be queried. This system was widely used as the basis for other NLIDB since its code was widely circulated.

- ASK [3] had its own built-in database, and was able to interact with external applications, namely external databases and e-mail programs. An interesting feature was that the system's users were able to add or *teach* words or concepts to the system during their interaction.

Some of these NLIDBs were made available commercially. However, since they needed to be designed and implemented specifically for a given domain and database, other generic interfaces, such as form-based interfaces and graphical interfaces, seemed more attractive.

Architecturally, the NLIDBs used today are still similar to the early ones. There are *rule-based systems*, that can also be based on *syntax* or on a *semantic grammar*. Most of these systems use an *intermediate representation language*. We will now provide a brief explanation of these terms and will provide more details of these systems in Section 2.3.

A *rule-based system* needs a set of rules to which the user's question, or parts of the user's question, can match to. The rules establish patterns and the system then looks through them and sees which rule

matches to the given user question. This method is called *Pattern Matching*. A *syntax based system* [1] syntactically analyzes the user's question and uses a grammar with all the possible structures of the user question. The grammar is used to created a parse [1] tree for the user question, and the parse tree is then mapped to an expression in the query language used by the database. In a *semantic grammar system*, the semantics of the user question are also taken into account and when the grammar categories do not match the syntactic constituents, the semantic grammar enforces constraints. This means that if a word has several interpretations in a syntactical perspective, but the word is not categorized semantically in more than one, a semantic grammar will only recognize the semantical interpretation. Consider the small example, with the following simple grammar, adapted from [1] in Listing 2.1.

Listing 2.1: Example of a Syntax based grammar. Adapted from [1].

S → **N V N**
N → "rock" | "magnesium"
V → "contains"

Both words *rock* and *magnesium* are classified as nouns in this *syntax based system*. That means, with this grammar, we can construct both sentences "magnesium contains rock" and "rock contains magnesium". A *semantic grammar system*, would have the semantic grammar in Listing 2.2 to complement the grammar present in Listing 2.1.

Listing 2.2: Example of a Semantic based grammar. Adapted from [1].

S → **question**
question → **specimen info**
info → "contains" **substance**
specimen → "rock"
substance → "magnesium"

Here, we can see that only the second sentence "rock contains magnesium" would be accepted by the semantic grammar, which is indeed the more accurate sentence.

Systems with an *intermediate representation language* transform the user question into an intermediate language that expresses the meaning of the user question in an internal language. This is meant to abstract the meaning of the user question in a way that is not specific to the database. The intermediate representation can then be expressed in the query language.

Around the same time NLIDB's started to appear (60's and 70's), another kind of NLP Systems was being created. They were called Chatter bots (the name was coined after the CHATTERBOTS [4]

[1] "Parse" can also be used as an action, parsing. It refers to the separation of a text, string or word.

system), and were meant to maintain a conversation with the user. A couple of the most popular systems at the time were ELIZA [5] and ALICE[2] [7]. ELIZA was meant to respond roughly as a psychotherapist, interact with the user, encouraging the user to communicate more, and express himself. Peter Norvig, in [8] details ELIZA's Pattern Matching implementation. ALICE had some information on how to deal with questions about its own consciousness and used Supervised Learning[3] to make its answers more believable. This was accomplished by monitoring ALICE's interactions and developing algorithms that could automatically detect dialogue patterns with no specific reply and creating that reply content.

As the years progressed, these so called Chat-bots (the name Chatter bots has been shortened in recent times) have diverged from just conversing, to responding accurately to questions. This has been accomplished with the use of NLP.

Most commercial websites nowadays [9, 10] have assistant bots, which try to answer the user's questions, before connecting the user to a human assistant. These NLP bots facilitate the interaction of the user with the commerce, since small chores can be done by the bot, namely looking up data in a database and confirming deliveries.

WolframAlpha[4], a search engine released in 2009, provides query answers by computing them from external data instead of displaying a list of documents. WolframAlpha computed mostly mathematical questions, but as of recently, it can also provide the altitude of Boeing 737-700, the amount of Vitamin A in sweet potato, statistics on murder and manslaughter in Phoenix and New York, words with a certain pattern, genealogical relations with genealogical tree demonstration, and it can compute iterations of various fractals. Even Google[5], has recently been computing the answers to several types of query, particularly mathematical calculations and biography searches. For example, if one were to search for model turned entrepreneur Karlie Kloss, Google would show biographic information, and social media and news appearances before showing movies she has acted in, direct links to her social media, and connections to Karlie Kloss that other people have searched for. Figure 2.1 shows the first half of this search.

Several group communication systems are also creating bots well integrated in the system such that it can interact with the users in NL while also providing relevant information. Slack[6] and Discord[7], already allow this sort of bot integration [11], as does Google's Allo[8].

It should be noted, that while years ago there was a clear distinction between Chat-bots and NLIDB, this is not the case anymore, and NLIDBs are now Chat-bots with tweaked response techniques.

[2] ALICE has had several re-implementations since the original implementation of 1995; however, Richard Wallace only wrote about ALICE in 2009. [6]

[3] Supervised Learning is a Machine Learning Technique that trains the system with data previously classified. The system then infers a function to map new data.

[4] http://www.wolframalpha.com/. Last accessed May 2018.

[5] http://www.google.com/. Last accessed May 2018.

[6] https://slack.com/. Last accessed January 2018.

[7] https://discordapp.com/. Last accessed January 2018.

[8] https://allo.google.com/. Last accessed January 2018.

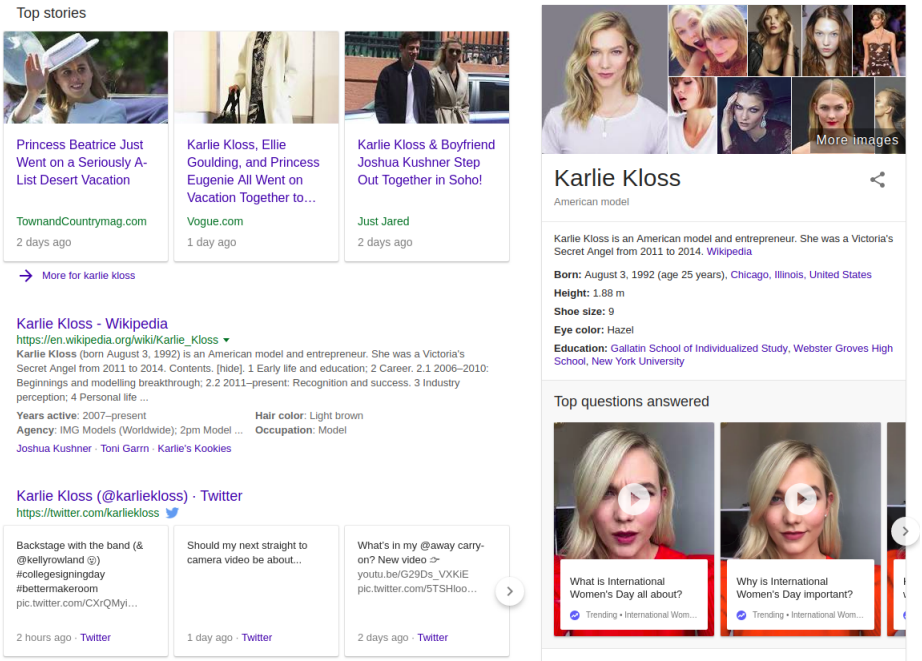

Figure 2.1: First results shown when searching for Karlie Kloss. This Figure contains biographic information, social media and news appearances. The movies she has acted in, and her direct social media links, also appear below these results. Last accessed May 2018.

2.3 Main Components

Figure 2.2 shows an overview of the most common Natural Language Interface (NLI)[9] components. The systems have the five major components mentioned in our Goals in Section 1.2: *Data Management*, *Question Analysis*, *Query Construction*, *Answering Step* and finally, a *Learning* component. We will now provide a detailed overview of these modules:

- **Data Management**

  NLI systems need to work on information. This information is where the system will gather the answer to the user questions. There may also be some *Vocabulary* specific to the context of the system, which can be accompanied by a glossary and a list of synonyms. The *Vocabulary* will help in obtaining the different possible meanings of the user question. If there are words in the user question that cannot be found in the information sources, the system can also use the *Vocabulary* synonyms to substitute those words and obtain a new question with the same meaning as the user question. This technique of using synonyms avoids a possible mis-comprehension of the user question by the system.

  This component encompasses obtaining, organizing and structuring the information whichever way

[9] NLI are interfaces where the user can interact with the system through NL questions or requests. These eliminate the need to know the exact syntax of what the request needs to look like for the system.

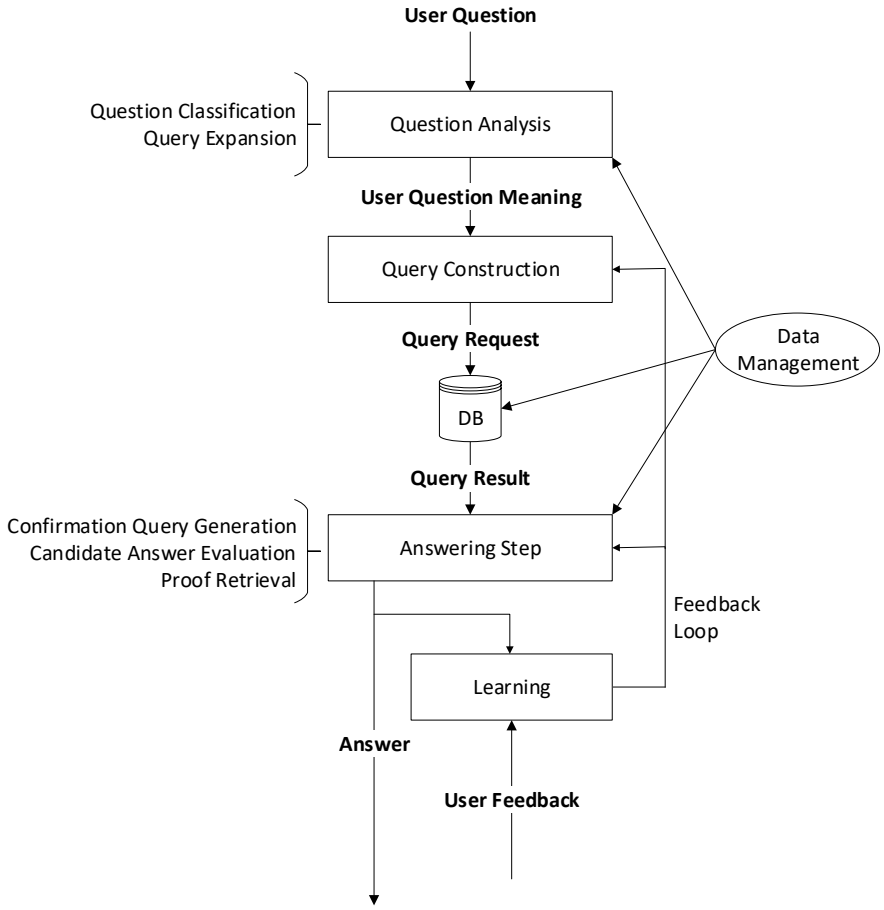

Figure 2.2: General overview of a NLIDB system.

is best. This structuring and storing can be in database, text files, or other means [12–14].

- **Question Analysis**

  The *user question* is received by the system. The goal of this component is to interpret that question, that is, to understand its meaning. If the system incorrectly understands the user question, the search will not be accurate and therefore, neither will the response.

  This is a NLP component, and several NLP tasks can be considered to analyze the user question. Some of the most common ones are *Question Classification* and *Query Expansion*. We will present an overview of these tasks.

  – Question Classification

    This task takes the user question, and associates it with the type of answer expected [15, 16]. This allows for a limited search and/or for a verification once the search results are back. For

example, imagine we have the user question "When was the deal with Client X signed?". This user question expects a *date* as answer. Having this information will allow us to exclude any other relations between the terms "When" and "deal", except for a *date* type.

Question Classification also benefits from *Headword Spotting*. Each user question has one or more words that can be seen as the most important, or most relevant, of the user question. This is the word allows the user question to be categorized correctly, and it is called the *headword*[10].

If there are certain patterns that can be found in the user's questions, the system may benefit from having fixed rules, that can be matched to, and prevent the system from needing to analyze the question from scratch. This is called *Rule Matching*. We may, for example, have a rule stating that when the words "What" and "year" appear in the same question, the Question Classification will be "date".

– Query Expansion

Query Expansion tries to find other ways the question can be formulated. This can be done in many ways, but we will mention two in particular:

1. Stemming the words of the question [15, 17]. This allows us to eliminate any affixes. We can then reformulate the query substituting the words with affixes by its simple word, with no affixes. If the headword of the question is known, then the query reformulation can focus only on the headword stemming instead of stemming every word of the query.

2. Finding synonyms. This avoids the search returning no results when the system only recognizes a limited amount of words. Some systems use outside sources for synonyms, like WordNet [11] [18, 19], while others create a system knowledge base that contain some synonyms or expressions accepted [12].

It should be noted that in Figure 2.2, the *User Question Meaning* is located after *User Question Analysis* and before *Query Construction*. This is because there are several systems that have an intermediate language [1], that represents the user question meaning, before passing the user question to the query language. Although not all the systems mentioned below have this intermediate representation, since we are providing a general overview diagram, it has been included. It should also be noted that there are systems that avoid all intermediate steps altogether and constitute what are called end-to-end systems. The system presented in [20] relates a task-oriented dialogue

[10] A headword can also be called *lemma* and it is the dictionary form, or keyword, for a given meaning or sentence.
[11] https://wordnet.princeton.edu/. Last accessed January 2018.

system that is end-to-end. However, these systems require a large amount of past user questions, which is not available in our project, and therefore, this approach will not be followed.

- **Query Construction**

  When the meaning of the user's question is obtained, a query needs to be formulated to be inserted into the system [17, 21]. The language this query is formulated in depends on the implementation of the system. The systems we will study formulate either a Structured Query Language (SQL)[12] or SPARQL[13] query.

- **Answering Step**

  The query is computed, and the system's response is evaluated. Most systems obtain a list of candidate answers and score them [13], while others consider only one answer at a time, and provide proof with each answer [22, 23]. There are also a few systems that perform a second query on the system, that consists of a combination of the answer and the question [24].

- **Learning**

  Finally, there is a feedback mechanism, that is fed back into the system to improve its results, as a learning method. The system may ask the user to confirm if the answer was satisfactory, or for instance, to rate three answers in order of usefulness, or even to provide a better answer [25, 26].

2.4 Data Management

Data Management is the component that retrieves, organizes and structures the information that will used by the NLIDB.

The DeepQA system [26, 27] is used nowadays in many Watson APIs, independently of the specific knowledge they use. DeepQA acquires all the information it finds relevant before the launch of the system, since it only functions offline. For that, the system analyzes questions that have been asked before, and pinpoints the types of information it should contain. It then acquires the data from encyclopedias, and other large information points. An *automatic corpus expansion* is done. This identifies and retrieves documents related to the relevant topics, extracts text from the data, scores the text relatively to the original data document, and merges the most informative text into an *expanded* or *baseline* corpus. This is all done in preparation for the system to run, since the *expanded corpus* is what the system will effectively use as its data source.

[12] SQL is a domain-specific language used to manage data stored in databases.
[13] SPARQL is a language to query databases by creating a semantic query.

On the other hand, we have other Watson systems that access the Internet, like the Semantic Web Search Engine. The Semantic Web Search Engine, [18], explores the content of the Semantic Web[14]. Its three main activities are collecting content, analyzing and extracting metadata and indexes, and implementing efficient query that facilitate accessing the data. This system works with keywords or with direct SPARQL query, but the data is extremely organized. The content of the retrieved documents is looked through and analyzed. The documents' data is then extracted so that it can be used. The *semantic documents* are characterized in terms of their content. Watson also uses unique identifiers, called internal identifiers, for each source document in case there is a mismatch in any Ontology[15].

When we have database or tables, we can use a technique similar to AskHERMES [24], a medical answering system. It has tables incorporated into the system's knowledge base, and to provide context and relation between the column and the entry, *the column name is added to the content of each table element*. For example, a table with a column named "Client Name", and with entries "James P." and "John M." will join the column name with the entries, and save the values as "Client Name, James P." and "Client Name, John M.". This process allows a clearer organization of the data without having to preserve the whole database structure.

2.5 Question Analysis

The systems detailed below have different methods of understanding what the *user question* means. The system will proceed and respond adequately only once the user's question is understood.

PRECISE, [17, 28], is a system that considers natural questions with implicit[16] attributes. When we communicate verbally, we usually do not take into consideration which attribute corresponds to which noun. However, if a written question is asked about a certain attribute, there is a need to understand what the question refers to. The PRECISE approach considers several steps to obtain a valid mapping of the question, and will then convert the mapping to an SQL query.

Its overall question analysis system is composed of a *lexicon*[17], a *tokenizer*, a *matcher*, and a *parser*. The *lexicon* performs two main operations: it can retrieve the tokens [18] that contain a word stem [19](when the word stem is provided), and it can retrieve the database elements that match a given token. To do

[14]The Semantic Web is an extension of the World Wide Web. The W3C (https://www.w3.org. Last accessed January 2018.) describes it as "common framework that allows data to be shared and reused across application, enterprise, and community boundaries" (https://www.w3.org/2001/sw/. Last accessed January 2018.)

[15]An Ontology is a set of concepts in a certain domain, containing their properties and relations.

[16]Implicit describes words that are not directly explained or mentioned in a sentence. That is, if we have the two sentences "Anna went to the movies. She got popcorn too.", the word "she" refers implicitly to "Anna".

[17]A lexicon refers to the vocabulary of a certain domain.

[18]A token is described as "a set of word stems that matches a database element" [17].

[19]A word stem is the root of the word, what defines the word, without any affixes.

this, the *lexicon* organizes the names of the database elements by extracting them and splitting them into individual words and stemming them. An ontology is used, to create a list of synonyms. Each element is then associated with its corresponding word stem and synonyms. The tokens point to hash tables that contain the database elements that they match to, and the tokens are also placed in a hash table indexed by word stems. This allows the *lexicon* to easily perform its two operations since the storage structure is built to accommodate them. The *tokenizer* stems the words in the user question. It then searches for the tokens which contain the word stem in the *lexicon*. The words of the question are in the tokenization. To obtain all the possible tokenizations of the question, it attaches to each token the types of database elements they can match to. The *matcher* is what is meant to solve the implicit words in the request. It finds a semantic interpretation of the ambiguous tokens as database elements. The *parser* parses the question and uses the parse tree to extract the attachment relationships between tokens.

Retrieving the word stems and synonyms helps with interpreting the different ways a question can be asked. Solving implicit arguments is also an important aspect of understanding the question, and relying on the semantic interpretation for that, ensures that the question understood is indeed valid.

MEANS [13], a medical question-answering interface, uses an interesting approach in its question analysis. Besides obtaining the expected answer type, the system also identifies the question's keyword as a *focus*, and any relation between this *focus* word and other mentioned entities are also obtained (if other entities are present). It also combines NLP methods that analyze specific corpora with Semantic Web technologies that offer both expressiveness and data sharing and integration. MEANS uses an ontology specifically built for the system, to represent the concepts and relations that are used to create the query translations of the questions. The MESA ontology also annotates corpora that is then used to search and extract the answers. Semantic Extraction[20] is done by analyzing the question to obtain relations between the entities mentioned. There are several relations that can exist between the entities, and they need to be considered separately. Another problem that needs to be considered is that the same relation can be described or requested differently. These relations are obtained with pattern extraction with regular expressions[21] involving keywords.

The questions are classified into ten groups, depending on the type and complexity that the answer will need. These classifications include "Yes/No" to "Definition" and "List". The main process consists of six steps:

1. Identification of the question type;

2. Determination of the expected answer type. This is done for wh-word[22] questions, which are tagged

[20] Semantic Extraction refers to the different techniques available in identifying and extracting certain words. These words can represent entities, facts and concepts.

[21] Regular expressions are a sequence of characters or symbols that detail a pattern to be found within a string or text

[22] Wh-word refers to the words "Who", "What", "Where", "When". These are the words most commonly used when forming questions, and are the words expected from a query. The word "How" is not always considered.

with a "1" but "Yes/No" questions are also tagged, with a "0";

3. Construction of a simpler form of the question;
4. Medical Entity Recognition on the question formed in the previous step;
5. Extraction of semantic relations on the question formed previously;
6. Construction of the query.

Besides just classifying the type of answer expected, NAUDA [29], takes into account if there should be an over-answer (over-answering will be touched upon in Section 2.7). NAUDA is inserted in the domain of water and waste management in a particular territory in Germany. This system classifies the question in order to understand the form of presentation the answer should have.

The default answer to the wh-word questions is provided in NL, as a sentence or a phrase, or as a simple value. If "Yes" or "No" questions are asked, instead of returning the simple short answer, the system uses over-answering [23].

NAUDA may also respond to questions related to particular areas of the territory, and, for that, they have decided to show topographical maps for the area in question. This is because some answers benefit from a *visual* aid rather than being only explained with words.

The proposed system in [30] translates the question asked into a First Order Logic (FOL) representation as an intermediate language. The FOL representation is passed to both SQL and SPARQL. The FOL is then translated into SQL and SPARQL. The system's grammar consists of grammar and lexical rules derived from interlinked database and knowledge base. There is then a disambiguation that relies on the features inserted for each prepositional phrase. These features determine the attachment these phrases will have. Undesired attachments are then filtered out.

DeepQA [26, 27, 31], includes *parsing*, *question classification and decomposition* and *detection of entities and their relations* as part of their question analysis. It will initially analyze the question and determine how the question should be processed. The question is then parsed in various forms, so that *semantic roles* and *relations* between words can be extracted.

Words with various meanings are considered at this point. *Question decomposition* happens here: if a word has two different meanings for example, the system duplicates the question and tests the question with both meanings possible. If both searches return a valid answer, the final answer will be the combination of both searches.

To classify the answer, semantics are not considered, but rather there is a search for a Lexical Answer Type (LAT). That is, for example, if the context of the question is "Chess", the domain of the search

will be limited, but will not provide any clue as to what the answer is, while if the context is "Names", then the context itself is a starting point for the system. As in the system [17], there may also be some words or simple relations that might need to be inferred.

The aim of Just.Ask, [32, 33], was to create a rule-based question classifier, based on detecting the most important keyword in a question, the question's *headword*.

Two methods are used to classify questions:

- **Pattern Matching:** Pattern Matching requires manually creating patterns that can then be matched against the incoming question. If a match occurs, the system then returns a question category.
- **Headword Identification:** The headword identification is done by a rule-based parser. The headword hypernyms are then looked at until one can be categorized.

To identify and extract the question headword, the system creates a parse tree of the question. The tree is then traversed top-down, in order to find the question headword. The system uses a predefined set of rules, to decide which node is or contains the head of the question. This process is done for all branches, until a terminal node is reached. The rules chosen are adapted to extract headword from questions.

The categorization of the headword is done by mapping the word into WordNet [23] using a set of heuristics to aid with disambiguation. WordNet has a lexical hierarchy between each word and a higher-level semantic concept, which in this case will represent the question category.

2.6 Query Construction

Most systems studied provide little to no information about their query construction. We believe this is because the main concern of NLIDB's is understanding the user question, and because once the meaning is obtained, it is expected that a database user (and the NLIDB creator) is able to create the query. Nevertheless, we provide information on one system that has a confirmation technique for the query created.

The PRECISE [17] system's query generator creates single-relation query, and if there is a need for multiple-relation query, then it uses the JOIN operation. If there are many valid mapping possibilities, then various query are created and all are submitted.

The single-relation query are created with a SELECT, WHERE and FROM. The SELECT contains the element that is connect to the wh-word of the question, the WHERE contains the attributes and values, and the FROM the relation name for the attributes mentioned.

[23] https://wordnet.princeton.edu/. Last accessed January 2018.

Finally, there is an equivalence checker. It is used when a matching algorithm returns that two query are equivalent. The equivalence checker then asks the user to confirm which query it should consider as the correct interpretation.

2.7 Answering Step

Investigating the answer means confirming that the answer provided to the user is indeed a valid answer (or is valid with a high probability), and the answer that the user desired. We will now walk through several ways of Answering.

Starting with the most simple, the AskHERMES system [24], *confirms the answer* by *restructuring the query with the answer*, and checks the provided answer with information from the system. For example, if the original question was "When was JFK assassinated?", and the answer obtained was "1963", the system would request a new query with "Was JFK assassinated 1963", and try to obtain a valid search. If the date was in fact correct, the search would be valid.

The DeepQA system [26] divides the search into phases. It performs a primary search, in its information sources, with the goal of obtaining as many potentially valid answers as possible. These potential answers are called *candidate answers* and can then be cross-validated with information obtained from the question itself.

These *candidate answers* are filtered, in a process called "Soft Filtering", which selects through all the initial set of candidates to create a smaller set. For a *candidate answer* to pass to this smaller set, it must pass several scoring algorithms, and have a score higher than a threshold. This threshold is obtained after having trained data with machine learning [24]. There is then a second scoring on over fifty components that is only done to the *candidate answers* that pass the first filtering. This stage is called hypothesis and evidence scoring. It consists of gathering additional information to support the candidate and applying scoring analytics to evaluate the evidence. One effective way this system gathers additional information is by doing just as AskHERMES [24]: to add the candidate answer to the original query and see if any results are delivered. The same technique is used differently in these two systems.

A different way of responding with additional information is by *over-answering*. *Over-answering* tries to avoid answering with "Yes" and "No" when the user may want more information. The system [23] points out five different ways of *over-answering*, namely *providing unsolicited justifications*, and *offering to provide information about a topic* as the system discovers it.

[24] Machine Learning englobes techniques that allow a computer to learn and act without being explicitly programmed.

The system can over-answer by identifying keywords such as quantifiers that could benefit from further explanation. Sentences with "Have/Has" can have a simple "Yes/No" answer, and usually have a follow-up question. The system avoids having to analyze and respond to a second question by looking into the system for *evidence* of the first question, by determining *truth values*, *quantifying the question*, and obtaining and computing a more *complex response*, with additional information.

We can also consider the idea that the answer provided to the user must be justified. In [22], the user is informed of what part of the submitted question led to what part of the answer. There are three key ideas to acquiring this information:

- **Provenance Tracking:** This tracking is done with two mappings. The first is accomplished by noticing which parts of the NL question translate to which parts of the formal query. The second is by evaluating the query using a "provenance-aware engine" which stores which parts of the query contribute to which parts of the answer provenance. These two mappings, designated "text-to-query" and "query-parts-to-provenance" allow the system to infer which parts of the text relate to which provenance.
- **Factorization:** The idea is to reduce the provenance size, since one answer can come from various explanations. The factorization is meant to group similar or equal explanations.
- **Summarization:** The third idea is to summarize the results of the factorization. This leaves each explanation with a short summary of what it is about. While there is a small loss of information, the information becomes easier to follow. Each factorization can be summarized in many ways, so the system enables summarizations with different granularity levels.

Another way of proving why the answer was selected is to evaluate it. MEANS, [13], uses Mean Reciprocal Rank (MRR), Precision[25], and Recall[26] to evaluate the answer that is to be provided to the user. The system then returns a limited and ordered list of proposed answers to the question asked. The answers take into account two aspects: the granularity and their justification. As we have seen, justifying the answer will allow an easier confirmation or denial of the proposed answer.

For the MEANS system to be useful to doctors, the time the search takes needs to be as minimal as possible. Their research showed that the search will be canceled unless it is completed in less than thirty seconds. We can relate that to our daily searches on Google, and how we as its users usually do not go past the second page, and prefer changing our query terms.

[25]Precision is the ratio between the relevant selected answers and the selected answers. It can also be seen as obtaining the actual number of true positives amongst both true positives and false positives. Imagining that we have a system with ten answers, five of them relevant. If the system selects six answers, and only two of them are relevant, its precision is 2/6.

[26]Recall is the ratio between relevant selected answers and total relevant answers. It can also be seen as obtaining the true positives amongst both true positives and false negatives. Imagining the same scenario mentioned above in *Precision*, the recall of the same case would be 2/5.

2.8 Learning

We can improve the Precision and Recall of a system with its learning abilities. These are what allow it to remain useful and up-to-date. There are many different approaches when it comes to learning, but in the context of this project we focus on learning methods that take user feedback into account. This way, the users can provide their opinion on which answers were actually relevant to their requested question. As mentioned in Section 1.2, we can use feedback in different ways, such as improving the ranking algorithm, or adjusting the weight of certain answers.

There is a patent from 2005 [34] that describes several ways of applying user feedback to database searches. The context of this patent is different than the one we are considering. The patent considers accessing the database records directly. However, here are the main ideas:

- If the user accesses a record in the result set, and that is the last record accessed, then that means that the user has found the answer he was looking for;
- For a given result set, a certain set of keywords may provide the correct record. If this set of keywords can be identified, they can be given a more relevant weight compared to other words present in the result set. If certain keywords in the user question always require a certain record, the same principle applies. This is an idea similar to *Rule Matching* but applied to the question-answer relation.

The system described in [25] works with user feedback. It has adapted neural sequence models to map the user questions directly to SQL. The idea is to skip any intermediate representation, and to have the final feedback come from crowd workers. The crowd worker can then mark the result of the mapping as correct, or incorrect. If the crowd worker considers the mapping incorrect, he will then be asked to annotate the mapping with the correct query. The correct mappings are then added to the training set.

DeepQA [26] does not require feedback, and instead scores the answers. The *candidate answers* are first filtered and scored on more than fifty components. After merging compatible answers, they are ranked and given a confidence level. The ranking and estimation of the confidence level is done by a machine learning approach which uses known answers as its training set, and then creates a model based on the scores. Finally, only the answers above a certain confidence threshold are presented. With this technique, only the answers that are considered more appropriate are presented.

Lastly, we will mention SEMPRE[27] [35]. It is a toolkit that allows the creation of semantic parsers,

[27] https://github.com/percyliang/sempre. Last accessed August 2018.

using a rule-based grammar. Once the grammar has been implemented, when SEMPRE runs a question, it attributes a score for each possible matched rule by adding a weight to the Named Entity Recognition (NER)[28] and Part of Speech (POS)[29] present in the questions. Each possible rule also has a probability of being the correct rule, that is, the one that should be matched. This probability is based on the score mentioned above, but takes into account a "temperature" intended to normalize the values. If the probability of the rules that can be matched is equal, the system choses one at random amongst them.

One of SEMPRE's components is responsible for learning from its rule-based grammar. The toolkit allows two methods of learning: supervised learning using examples, and using input from the user in runtime. To learn from examples, once the grammar has been implemented, another file is created containing examples that consist of pairs of utterances and their expected values. These examples then influence the score and therefore the probability of a certain rule being chosen in the future.

The other option is learning during the execution of the toolkit. This method requires the user to command the system to choose the rule intended. It is most useful in the case of rules with the same probability, where the user can clarify which rule it required. For future uses of the system, the user can load previous learnings in a new execution.

2.9 Summary

In this chapter we have looked at the history of NLIDB, and how there are different ways of implementing one. We have looked at *rule-based systems*, their requirements and types. We have also seen how a *Syntax based grammar* differs from a *Semantic based grammar*.

We have determined the major components in NLI, and what each of them is responsible for. The *Data Management* component deals with the domain of the system, and gathers the information that can be used by the *Question Analysis*. It is also in *Question Analysis* where NLP tasks are performed, such as question classification where the answer type is predicted, and expansion of the user question to gather its synonyms. The query is created in *Query Construction* and the answer for the query is obtained in *Answering Step*. The implementation details of these two components depend on the database or database table and the technologies in use. NLIDBs can also have a *Learning* component, with feedback mechanisms such as user input.

We can now create our implementation's pipeline based on what we have learned and on what components are useful for our specific domain, since there is no complete system that fulfills our requirements. We will look at our domain and what components we will implement in the next chapter.

[28] Named Entity Recognition locates entities in text and classifies them into categories.
[29] Part of Speech creates categories of words with similar grammatical and syntatical properties.

3

Pipeline

Contents

3.1 Introduction

Based on the work of the previous chapter, we made several decisions for this implementation. Firstly, for our *Data Management*, we chose to create named entities and synonym lists. Our *Question Processing* will then remove stop-words[1], and perform NER and synonym substitution.

Since we do not have many examples of the questions that the users would submit we could not justify pursuing an approach using machine learning, and therefore turned to a rule-based implementation. Rule-based implementations have been described in Section 2.2, as have Semantic Grammars, which we will also be using. We have taken into consideration the limitations of these decisions, and they will be mentioned throughout Chapter 3 and Chapter 4.

We have considered two approaches, a *semantic grammar approach*, from hereafter called sys1, and a *rule-based approach using the SEMPRE toolkit* (mentioned before in Section 2.8) *with a learning capability*, called sys2. Each of these approaches receives the processed question (from the *Question Analysis* step), and creates a query with the rules it matches to (*Query Construction*). That query is submitted into a BNP Paribas table, and the final answer is then provided (*Answering Step*). Our *Learning* is implemented as a component of sys2.

This chapter will focus on the process of creating the NLIDB. Section 3.2 details the domain of the system. Section 3.3 contains information about how the system functions, detailing the work done before and after a question is submitted to the system. Section 3.4 explains sys1 while Section 3.5 explains sys2. Please note that all data examples in the next chapters have been modified to ensure BNP Paribas data privacy.

3.2 Domain Characterization

To create a NLIDB, we need to understand its domain. As previously said, this NLIDB will obtain its answers from the table called Rollover Opportunities. This requires understanding the type of questions that will be asked, the questions themselves, where the answers can be obtained, what the answers are and what is the best way to acquire them. For that, a collection of ten questions, and documentation was provided by the GCA group. Part of this documentation is in Annex B. The table has a corresponding visual representation, called a dashboard, which shows some aspects of the table.

We will detail the table (and corresponding dashboard) in Section 3.2.1; the questions and their type will be explained in Section 3.2.2.

[1]Stop-words are words that do not contribute to the meaning of the sentence for a specific context, and can therefore be filtered out in NLP.

3.2.1 Table

The *Rollover Opportunities* table provides information regarding upcoming trade expiry over the next 21 days for Foreign Exchange (FX). The dashboard (Figure 3.1) is used mainly by sales and traders, and allows them to make timely communication with their clients, so that they can continue to trade (rollover) with BNP Paribas.

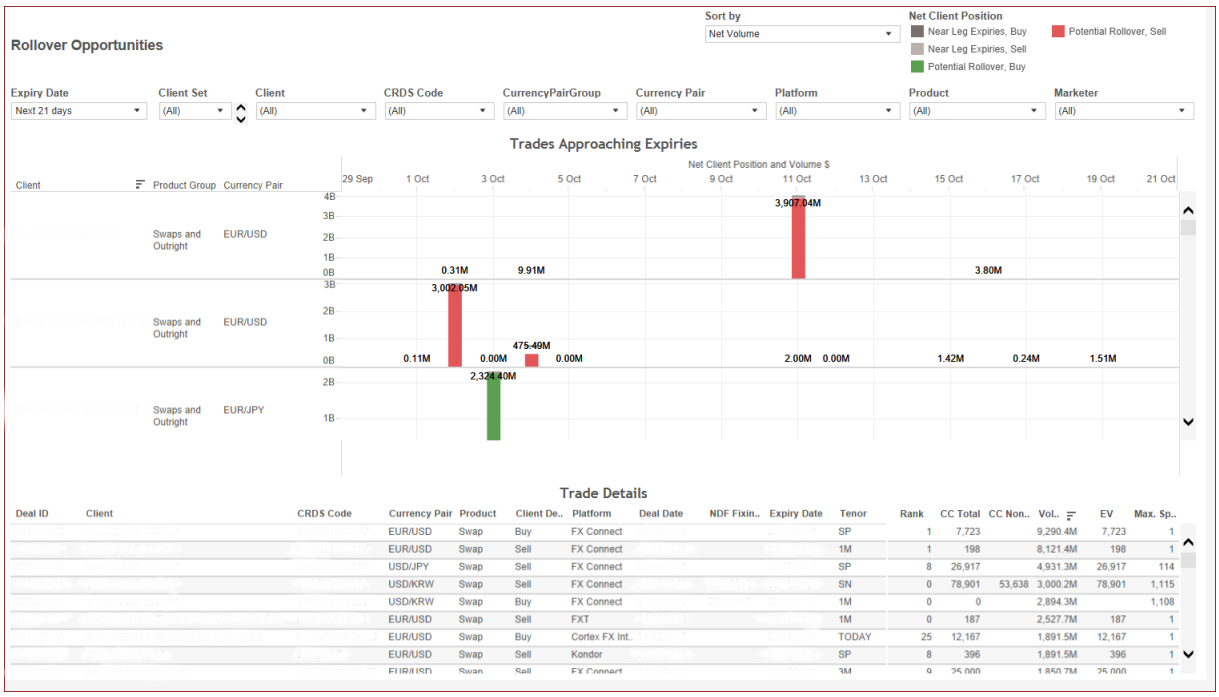

Figure 3.1: Screenshot of the dashboard in use. Client names, codes and dates have been removed.

The fields of the table are split internally by BNP Paribas into two categories: ***Measure*** or ***Attribute***. We have for example, *Client* (a string), and *Expiry Date* (a date) defined as ***Attribute***, while *CC Total* (a number) and *Volume* (a number) are ***Measure***.

A complete list of this division is presented in Annex A. This division is relevant since we can use these categories to create our grammar and generalize rules.

3.2.2 Questions

The questions that can be asked in this dashboard use both types of fields mentioned above in Section 3.2.1. There can be questions regarding one or two *Attribute* fields, while *Measure* fields can only be questioned in a simple question or in regard to other *Attribute* fields. To answer questions such as the ones presented in Listing 3.1, we can combine the different fields into the grammar sample rules presented in Listing 3.2.

Listing 3.1: Example of questions that can be asked by the user. The Measures and Attributes present in the questions are underlined.

"What is Client John Doe Volume?"
"What is trade t567uio9 cc total?"

"What is trade th4ksw1 currency pair?"

"Which is the deal id of client Mary Oakley?"

Listing 3.2: Structure of some questions that can be asked by the user.

attribute VALUE **attribute**
attribute attribute VALUE
attribute VALUE **measure**
attribute measure VALUE

If we revisit the fields mentioned in Section 3.2.1 (*Client*, *Deal ID*, and *Expiry Date* are ***Attributes***, while *CC Total* and *Volume* are ***Measures***), we can observe how the four questions in Listing 3.1 belong to the types mentioned above. Please note that exact values from the database table have been replaced, and that the questions presented have not been stripped out of the stop-words and wh-words to allow for a better understanding of the questions.

Besides these four basic rules, we also have to include other types of questions, namely excluding entries with values from a certain ***Measure*** or ***Attribute*** field, and obtaining the top value of a ***Measure*** or ***Attribute***. An example for each case is present in Listing 3.3. In this domain, obtaining the lowest value is not of interest, since the user will want to maximize the values to rollover. The database table has information about expiring dates, and the user will want to acquire information about the nearest expiring date, or the ***Measure*** that will be higher and provide more value when rollovered.

Listing 3.3: Example of questions that can be asked by the user. These questions show the cases of *top values* and *excluding* a certain value.

"Which client has the highest Volume?"
"Show me all platforms excluding plat09"

We could also combine a ***Attribute*** with several ***Attributes***, or with several ***Measures***. We have however, focused on rules that will match to questions using one or two fields of the types ***Attribute*** and ***Measure*** with one or two values known. This decision comes from the fact that a complex question (one with several ***Attributes***) cannot be generally explained by a rule. Since our system is rule-based, this causes a problem. This is a topic of *Future Work* and mentioned in Section 5.3.

Going into the next sections, there are a few considerations we want to point out. The different wh-words used are not a limitation to the system since the core of our NLIDB does not use these words to obtain the meaning of the question. We have also included a question in Listing 3.3 using the words "Show me" which is a synonym considered. Section 3.3.2 will go into further detail.

3.3 Proposal Overview

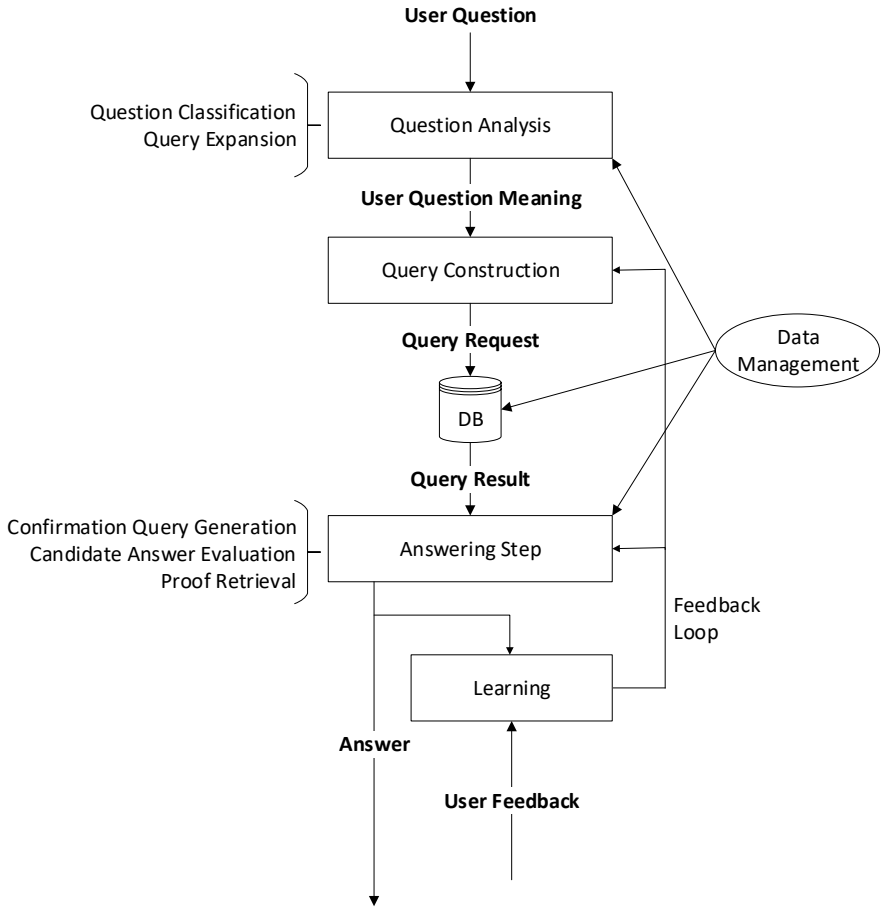

Figure 3.2: General Architecture provided in Chapter 2.

After researching NLIDBs in Section 2, we had created a general pipeline. That pipeline is replicated in this section in Figure 3.2 for ease of comparison with the pipeline implemented, present in Figure 3.3. Our new pipeline implements the five components identified in the general pipeline. The main difference is the way *Data Management* and *Question Processing* interact in the whole system. Our *Data Management* interacts with the *Question Processing* unit, as well as with the table and the sys2 grammar. Our *Learning* is also within sys2 instead of interacting with other components. We will now

further detail the pipeline.

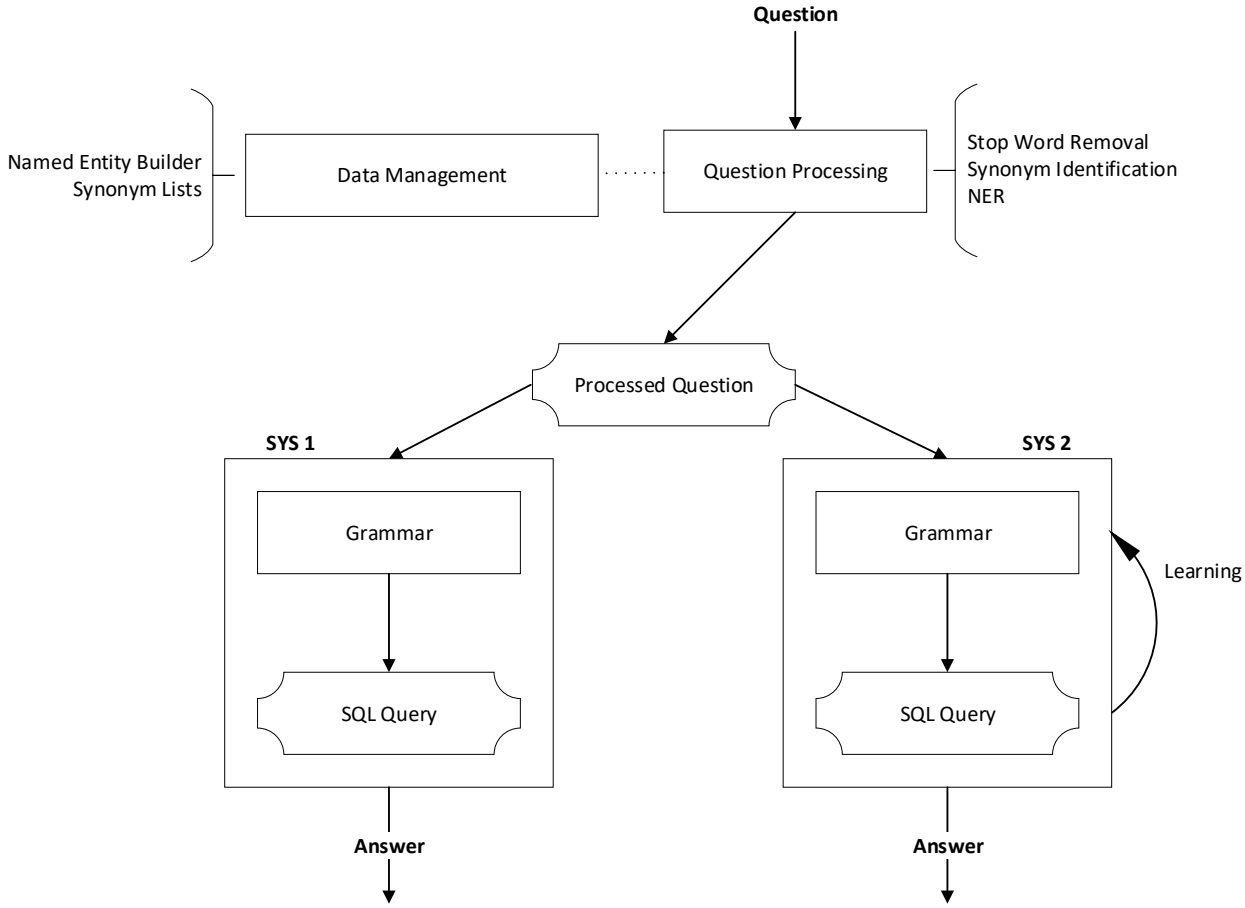

Figure 3.3: Architecture implemented.

The first component, *Data Management*, is called before the questions are received. This includes retrieving information from the database table to create categories (Section 3.3.1), having a list of synonyms and making sure the sentence can be sent to the rules and will not create a rule that cannot be queried (both Section 3.3.2). After the system receives the question, and processes it (**Processed Question**), both approaches implemented are called separately. Section 3.4 will detail sys1, while Section 3.5 will detail sys2, including its learning component.

3.3.1 Data Management

When a user submits a question, the information submitted may not be correct. They may have misspelled a word, or requested an entry that does not exist in the table. To avoid this, we query the

information from the table using the query structure in Listing 3.4, and create local lists of the table entries to perform *NER*. We have defined[2] the categories based on their field type, and the question structure mentioned in Section 3.2.2. When the question is submitted, these values can be compared with the ones present in the user question, to confirm they are valid for the table in question. This avoids the blockage of the question further in the pipeline since a query that cannot be answered will not be created by the system. For example, if a user submits the question "When will deal 78560 expire?", the system will confirm if the deal id 78560 exists in the list of deal identifications. If it does not, it will inform the user that the values submitted are not valid. If it does, it will proceed to the grammars.

Synonym lists have also been created as part of this component, but since they are static and used as part of the *Question Analysis*, they are described in the next Section, Section 3.3.2

Listing 3.4: Query to obtain the values from the table.

```
SELECT *Measure/Attribute FROM table
```

3.3.2 Question Analysis

To process the questions, we pass the question through several steps: removing stop-words, such as "is" and "that", check for the existence of synonyms, and NER is performed as described in Section 3.3.1.

In an attempt to clear out the question as much as possible, we have studied the table and the type of questions to be submitted, and since the type of answer required can be identified by the table fields present in the question, we can also remove the *wh-word* present in the question. This is not a common procedure, since in most NLP tasks, you need the *wh-word* to identify the type of answer. However, the type of data of each table entry of a certain field is the same. If a client wants to obtain the expiry date of a certain deal, the question can be "When will deal x expire?" or "Which is the expiry date of deal x?". Now, the field **expiry date** is of the type **DATE**, so the *wh-word* used does not influence what type of answer the question requires, since it can be obtained from the rest of the question (in this case, from the words leading to the lemma **expire**). Looking at another example, if we have "What is the deal ID of client y?" and "Which is the deal ID of the client y?", they both want the field **DEAL ID** such that CLIENT = y. If we remove the *stop-words* and the *wh-words*, we will have "deal ID client y" for both of the questions. Removing the *wh-word* also allows us to reduce the possible paraphrases of a question, and, from an implementation perspective, allows us to use the same rule to treat both questions that originally had different *wh-words*.

[2]We looked at a matcher created by Spacy (https://spacy.io/. Last accessed October 2018.), but for us to train the NER with the categories necessary, we would need hundreds of examples. Since we do not have the amount of examples needed, creating a new NER within Spacy was not possible. Therefore, a similar approach was implemented without the use of the Spacy matcher library.

We have also created *synonym lists* to substitute words that can be said in different ways. The goal of these *synonyms* is to process the questions and limit the number of paraphrases that have to be considered. That is, it allows us to reduce the number of rules that have to be implemented to deal with paraphrases of the same question. If any of these words appear, they will be replaced, considered as the same word, and analyzed in the same way. Each *synonym list functions* as a category. If the word is in that category, it will be treated in the same way as another word in that same category, since we assume that they are *synonyms*. For example, if a question uses the word "expiration" while another uses "expires", they both belong to the "expire" category, they will both be considered as the word "expire". The word "expire" is the word that will be then present in the *processed question* going to the grammar. In total, we have 42 synonym lists. Some table fields do not have *synonyms*, while others have 9. Some of our *synonyms* are present in Listing 3.5. We did not use *lemmas* for acquiring our *synonym lists* because they would not take into account spaces, and synonym expressions, since *lemmas* only create variations of the word itself.

Listing 3.5: Synonyms for words used in the user questions.

```
expire_synonyms = ['expiration', 'expire', 'expires', 'expiry', 'end', 'terminate', 'expiration date']
exclude_synonyms = ['exclude', 'excluding', 'remove', 'no', 'not', 'different']
higher_synonyms = ['more', 'greater', 'higher', 'superior', 'surpassing', 'larger', 'above', 'highest', 'top']
crdscode_synonyms = ['crds code', 'crdscode']
client_synonyms = ['clients', 'client', 'client name', 'clients name', 'client id']
```

Another question processing aspect has to do with subjectiveness present in the question. In our case, the interface needs to respond to temporal subjectiveness. To deal with that, we have created[3] a series of regular expressions that deal with both subjective and objective date expressions. This means that the NLIDB can deal with expressions such as "yesterday", "last week", "last quarter", "2017" and "may".

Listing 3.6 has a list of the date expressions that are accepted.

Listing 3.6: Temporal rules.

```
i
```

When these expressions are present in the question, they are marked and the corresponding date is calculated. The date is calculated in two formats: days (that corresponds to the number of days between the current date and the date expression) and in the time format DD-MM-YYYY. We calculate the dates in these two formats because some fields of the table use one format, while others use the other. Depending on the field of the table present in the question, the corresponding format is then placed in the question to be read by the grammars. The date expressions are calculated in runtime, and sys2 needs

[3]Several systems were tested but none of them responded in the way needed for this project. As an example, we have https://github.com/bear/parsedatetime. Last accessed October 2018.

to have the rules declared before its grammar runs. To deal with this, we have a function that creates a rule with the necessary date expressions as soon as they are calculated. This way, when the grammar is run, the rule exists and can be matched to.

If there are currencies present in the question, they are also treated. As requested by the GCA group, if the user wants to know about a currency pair, both variants of the currency pair have to be considered. That is, if the user wants to query about the pair *USD/EUR*, the inverse, *EUR/USD* must also be considered. For this reason, whenever the NLIDB identifies a currency, two questions are created (one for each currency pair variant) and both questions will then pass through the grammars.

3.4 Sys1

We have created our grammar using a *lexer*[4] and *parser*[5] called PLY[6]. The entire grammar is in Annex C, but we have included a snippet of the *lexer* in Listing 3.7, and of the *parser* in Listing 3.8. Our parser has a total of 15 rules, with 6 non-terminal symbols, and 41 defined terminal symbols, while there are non specified terminal symbols that correspond to the VALUE and DATE.

The first thing to declare in the *lexer* were the tokens that we would be using. These are ***Measure***, ***Attribute***, ***Exclude***, ***Value*** and ***Measure***. The first two tokens were to declare the possible ***Measure*** and ***Attribute*** names mentioned in Section 3.2.1. ***Exclude*** and ***Measure*** were to deal with the keywords referent to *exclude* and *higher*, that will be needed to answer the questions relative to higher values of a certain field, or excluding a certain field. These two cases were defined above in Section 3.2.2. The *Value* receives any alphanumerical token, which in this grammar will be the names of the entries the user knows, or wants to obtain information about.

Listing 3.7: Main Components of the Lexer of the grammar for sys1.

```
tokens = ('Measure', 'Attribute', 'EXCLUDE', 'VALUE', 'MEASURE')

t_MEASURE = r'volume | totaltrades | stddevrolloverdays | spot \s price \s strike | rolloverratio | rank
| netvolumeratio | netvolume'
            r'nearlegvolume | latestrolloverdays | ev | cc \s total | cc\s nonrisk | cc \s atrisk | averagerolloverdays
| maxabsnetvolume'

t_ATTRIBUTE = r'tradestatus | tenor | product \s group | product | platform | newtrade | netclientposition
| neartenordays | '
            r'maxfuturedate | marketer | localblotter | leg | fromdays | fartenordays | expiry \s date
```

[4]A lexer is the process of converting characters into tokens with assigned meaning.
[5]A parser takes data (in our case, the tokens created by the parser) and builds a structure while checking for a predefined syntax.
[6]https://github.com/dabeaz/ply. Last accessed August 2018.

```
| deal \s id | deal \s date'
                r'| currencypairgroup | currency \spair | crdscode | client \s deal \s side | client'
                r'| broker_fxt | ndf | fixing \s date'

t_VALUE = r'([a-zA-Z]{3}[/][a-zA-Z]{3})|[a-zA-Z_0-9]+'
t_DATE = r'([0-9]{4}[-][0-9]{2}[-][0-9]{2})|([0-9]{9})'

def t_EXCLUDE(t):
  r'exclude'
   return t

def t_MEASURE(t):
  r'highest'
  return t
```

As can be seen in Listing 3.7, ***Exclude*** and ***Measure*** are defined differently than the rest of the tokens. That is because the *lexer* matches the tokens in order (the longer rule with be tested for a match first), while definitions are considered before the token rules[7]. Since we want the words that refer to these terms (*exclude* and *higher*) to match to their unique rules instead of matching to ***Value***, we have declared them as definitions.

Listing 3.8: Part of the yacc of the grammar for sys1.

```
def p_statement_single(p):
  ''' statement : Measure VALUE
                | Attribute VALUE '''
  p[0] = ''.join(('SELECT *FROM table WHERE ', p[1], ' = ', p[2]))
```

As for the *parser*, we define each of the statements in a similar way as the one provided in Listing 3.8. The statement shown accepts a ***Measure*** followed by a ***Value***, or a ***Attribute*** followed by a *Value*. We then define what the **p[0]** will be, that is, what the statement will return. In the case shown, it will be the SQL query SELECT * FROM DB WHERE 1 = 2 where 1 will be the ***Measure*** or ***Attribute*** matched, and 2 will be the ***Value***.

3.5 Sys2

We wanted to test another approach, and explore how a semantic grammar could work with a learning component, and for that we used the SEMPRE toolkit described in Section 2.8.

[7]http://www.dabeaz.com/ply/ply.html. Last accessed August 2018.

The SEMPRE toolkit uses a grammar, that will be explained in Section 3.5.1, which is trained by examples, from which the toolkit provides a score and the probability for each rule the question matches to. Section 3.5.2 will focus on the learning component.

3.5.1 Grammar

The implemented grammar is as general as possible without compromising the result. This way, if new vocabulary needs to be added, depending on the vocabulary in question, there is the possibility of changing an already existing rule instead of creating a new one. The complete grammar can be seen in Annex D.

The grammar implemented contains several types of rules. There are *string declarations*, which declare table entries, ***Attributes***, and ***Measures***, as presented in Listing 3.9. These rules need to have their name without spaces (*nonriskcc* instead of *non risk cc*), and because of this, when the pipeline is run, two questions need to be created. As the pipeline substitutes the synonyms and named entities, the question that is created for sys2 with the replacement words has into account this limitation, and writes a different word for the sys2 grammar than for the sys1 grammar.

Listing 3.9: Measure declaration rules defined in SEMPRE toolkit.

```
(rule $Measure (volume) (ConstantFn (string "Volume")))
(rule $Measure (cc) (ConstantFn (string "CC Total")))
(rule $Measure (nonriskcc) (ConstantFn (string "CC NonRisk")))
(rule $Measure (atriskcc) (ConstantFn (string "CC AtRisk")))
```

There are also *general rules* that encompass the whole question asked, using lambda calculus[8] such as the ones in Listing 3.10.

Listing 3.10: General rules defined in SEMPRE toolkit.

```
(rule $General ($Measure $VALUE)
  (lambda m (lambda t (call + (string "SELECT *")
    (string " FROM table ")
    (string "WHERE ") (var m) (string " = ") (var t)))))
```

For instance, the rule in Listing 3.10 allows the user to ask questions about the value of a certain client's volume for example, or cc total, which are both declared in Listing 3.9 as ***Measure***.

The SEMPRE grammar rules function as part of a tree as can be seen in Figure 3.4, starting with the rule present in Listing 3.11.

[8]Lambda Calculus is an abstraction computation that uses variable binding and substitution.

The process starts by matching the *ROOT* with one of the other *GENERAL* rules. Since the *GENERAL* rules use variables, these variables are then matched with the declaration rules, providing the complete rule through substitution. The SQL query is then returned.

Listing 3.11: SEMPRE Root rule.

```
(rule $ROOT ($General) (IdentityFn))
```

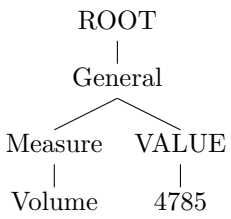

Figure 3.4: Tree for word matching given the rules provided in Listings 3.9, 3.10 and 3.11

3.5.2 Learning

The SEMPRE toolkit has a learning mechanism, to improve which rules it matches to. It has two learning approaches, which have been mentioned in Section 2.8. We have used the approach based on examples that are trained before the system runs.

To train sys2, we used examples containing a question and answer.

Sys2 matches each input to the available rules. Each valid rule will have a score, and a probability as detailed in Section 2.8. If more than one rule match, it will base its decision on the probability of each rule being the correct rule for that case. If all possible rules have the same probability, then the rule will be selected at random.

We have created a file with *examples* for the rules, having a total of 24 examples. We have focused on *examples* for the cases where there can be ambiguity, and therefore there is more than one rule available to be selected. Each *example* is comprised of a *question* and an *answer*. The questions in this *example* file are not complete, since SEMPRE will never receive the original question but instead receive the question after having gone through the processing presented in Section 3.3.2. The *question* for each *example* is therefore equivalent to the one SEMPRE will receive for its grammar application. The *answer* provided for each *example* is the SQL query that is expected that SEMPRE will return.

A sample of this file is present in Listing 3.12, and the complete file is available in Annex E.

Based on the examples, as mentioned in Section 2.8, when the test criteria are run, the system increases the probability of the rules that fire for that question. The learning can then be loaded in a later iteration of the system, and the changed rule with the most probability will be returned.

Listing 3.12: Example of SEMPRE learning file.

```
(example (utterance "client x volume?")
    (targetValue (string "SELECT Volume FROM table WHERE Client = x" )))
```

3.6 Comparing Approaches

Since we have implemented two different approaches, we will now compare the positive and negative aspects of each of the approaches in Table 3.1. We will look at the results of each approach in the next chapter.

sys1	sys2
+ No need to learn how to use a logical language.	– Needs knowledge of the logical language.
+ Ready to go when it runs.	– Needs to be trained any time a new rule is added.
– It will not improve or learn with the users.	– + Allows learning.

Table 3.1: Comparing the pros and the cons of implemented approaches.

3.7 Summary

We have described how we implemented our system's pipeline, starting with the table's *information treatment*, going through the *NLP* and the implementation of two *rule based approaches*. One approach, sys1, has a semantic grammar to create the SQL queries.

Sys2 has a learning method besides its grammar that allows possible ambiguities to be resolved. We have used one of its two learning mechanisms by learning from examples.

We finished this chapter with a small comparison of the two rule-based approaches. We will now discuss the results of each of the approaches in the next chapter.

4

Evaluation

Contents

4.1 Introduction

The evaluation of our NLIDB consists of confirming whether the answers provided by the system are the answers wanted by the user.

We should note that this NLIDB does not directly provide the answer to the user question but rather provides the SQL query that once submitted to the table should provide the answer the user wants. Therefore, we will be evaluating the SQL query, since if the query created is correct, so are the results. We should also mention that, as in the previous chapter, the exact token values used for the tests have been replaced so as to not compromise the privacy of the GCA table and users. Section 4.2 will explain how the evaluation was implemented, Section 4.2.1 will detail the corpora used for the tests, and Section 4.2.2 will detail the measure used for the evaluation. The tests performed are described in Section 4.3 and Section 4.4. We conclude this chapter with some considerations about how the domain of a NLIDB influences its implementation in Section 4.5.

4.2 Experimental Setup

Although we have implemented two approaches, both approaches run simultaneously, with the pipeline calling both grammars, each with their own question. As seen in Section 3.3, the pipeline contains the same NLP treatment, but calls each grammar with their own question since the grammars are built differently. These differences are because of the SEMPRE limitations (mentioned in Section 3.5.1) and because of problems that appeared since the table fields are not normalized (some have _ while others use spaces or uppercase letters). Sys2 also requires that the server is called before the NLIDB is run. This server allows the system to extract the sys2 answer in JSON so that it can be returned to the pipeline.

4.2.1 Corpora

To create a corpus that would try to cover the whole range of questions that could be asked by the user, we developed a set of questions throughout the project. Considering a rule that requires a ***Measure*** and an ***Attribute***, we have included several questions, with different combinations of ***Measure*** and ***Attribute***, to ensure that all the table fields are valid. We have a total of 100 questions in the development tests, and 21 questions in the user test.

4.2.2 Evaluation Measure

To evaluate our system, we decided to use *accuracy* since we want to classify if the NLIDB's response is correct or not. We consider an answer correct if the SQL query returned by each approach, sys1 and sys2, is the expected query.

There are a couple of factors that will affect the *accuracy* of our NLIDB. If we have not considered a certain question, there will not be rules to deal with it, and therefore, the NLIDB will not provide a correct answer. If the rules implemented do not take into consideration all the possibilities of expressing one question, the user question will not return a correct answer. If the user question contains a synonym not considered in our implementation, the question processing will refuse the question and it will not go on to the grammars.

4.3 Development Tests

During our development, we conducted a series of tests consisting of 100 questions. Some of these questions have two words, while others have ten. The questions cover the whole grammar, and there are multiple questions per grammar rule. The development tests include variations of the same question, as can be seen in Listing 4.1, as well as combinations amongst the ***Measure*** and ***Attribute*** fields, as seen in Listing 4.3. The rules that match the questions in Listing 4.1 questions are in Listing 4.2.

Listing 4.1: Development questions used for testing, with variations on the question structure.

"what is **deal** 456 **client name**?"
"which is **client** Joao's **deal id**?"

"which are the **clients** present in **platform** x?"
"which are the **clients** that are not in **platform** x?"

"what is **ev** for **tenor** y"
"**ev tenor** y"

Listing 4.2: Rule matches for questions in Listing 4.1. We can see how different questions can match to the same rule, and how similar questions can match to different rules.

ATTRIBUTE: deal id VALUE: 456 **ATTRIBUTE: client**
ATTRIBUTE: client VALUE: Joao **ATTRIBUTE: deal id**

ATTRIBUTE: client ATTRIBUTE: platform VALUE: x
ATTRIBUTE: client EXCLUDE:not **ATTRIBUTE: platform** VALUE: x

METRIC: ev ATTRIBUTE: tenor VALUE: y
METRIC: ev ATTRIBUTE: tenor VALUE: y

Out of 100 questions, sys1 obtained an accuracy of 87/100 (87%), while sys2 had an accuracy of 88/100 (88%). The NLIDB created queries for 8 more questions, but the query created was incorrect,

while the other questions remained unanswered.

Listing 4.3: Development questions used for test, with variations on the table fields used.

"who is the **marketer** for **deal** 456"
"what is **deal** 456 **local blotter**"
"what is the **rank** of **client** joao"
"which is **product group** for **tenor** y"

The incorrect questions were due to problems with the word interpretation. There were cases where the synonyms matched were the wrong ones: in cases where the table field name has spaces, and there is a similar table field name without spaces, the NLIDB matched the field name to the one without spaces. As an example, we had a question that was to match to the ***Measure*** *NetVolumeRatio*, but instead matched to *Volume*. The words used in this questions were "net volume ratio", which are in the list of synonyms for *NetVolumeRatio*. However, if we had used "netvolumeratio" in the question, the match would have been correctly made to *NetVolumeRatio*. This situation happened in 3 cases.

There was 1 question where sys2 returned the correct query while sys1 did not return one. This is a strange case, which refers to the rule **ATTRIBUTE EXCLUDE TOKEN**. Sys1 is able to return a query in other questions of the tests that match to the same rule. We believe this is a bug in the system, but have not been able to identify it at the time of this delivery.

There were 8 other questions where sys2 returned a wrong query while sys1 did not return a query. This wrong query happened for the same reasons as above. The name of the ***Attribute*** was inserted with spaces, and the NLIDB matched it to a wrong field. When tested without the spaces, the system was able to provide an answer with both sys2 and sys1 .

As for the remaining questions, neither system was able to create a query. This is due to the *synonyms* and *named entities* created. The words used in the question did not match the words in the lists, and therefore, the system did not send the questions to the grammars. This means that both *synonyms* and *named entities* can be extended, to avoid these errors.

4.4 User Tests

For our *user test*, we asked the GCA team to write some questions that would be submitted to the system in a normal usage. In a first use, we obtained 14 questions which demonstrated what type of questions a user would submit.

Out of the 14 questions, we obtained an accuracy of 5/14 (35%) with both approaches. The rest of the questions could not be answered for various reasons. 3 questions did not match to any rule, while 4 questions did not have the vocabulary needed. For the questions that did not match, since none could be reformulated into questions that could match, they can be taken into consideration and added to the

grammar in the future. The *type of field* also needs to be stated before the value of the field, and this did not happen in 2 questions. We can see an example in Listing 4.4 which shows two problems: the first question does not declare the field type of *ti ago*, which is *Client* and also does not use the correct entry of the field, which is the entry declared in the named entity (in this case, the entry is *tiago*. The second question shows how the user question should have been asked so that the NLIDB would recognize the question.

Listing 4.4: Comparison of two questions, the first submitted by the user, unanswered by the system. The second, shows how the question should have been so that the system would be capable of answering.

```
"what is the CC for ti ago"
"what is the CC for client tiago
```

These user questions show that the user needs to know how the system works, particularly how the questions need to be expressed so that the answer will be correct.

Because this NLIDB is to be used repeatedly by a small number of people, these small details can be learned by the user. To confirm that, we ran a second test, for 7 new questions. Out of these, 5/7 (71%) were correctly answered by the system, while the remaining 2 had no rule to be matched. However, if the questions were reformulated, there would have been a rule matched. These questions were similar to "top clients with rank", when the system would only accept the question as "rank of top clients". This is also a case of a rule that can be added into the grammar in the future, but if the users know that reformulating the question leads to an answer, they may also do that in future uses.

4.5 What if we change the domain?

At the start of this project, the table we would be creating the NLIDB for was a different one than the one explained in Section 3.2.1. The table was called *Missed Voice Deals*, and it had the purpose of registering what caused the traders to miss a deal negotiated via voice. Some of the questions for the *Missed Voice Deals* table are present in Listing 4.5. Since it was still in development and it would be some time before users would have access to the dashboard and its table, the GCA group changed the table for this project to the one described in 3.2.1.

Listing 4.5: Example of questions that could be asked by the user in the Missed Voice Deals Dashboard.

```
"Which are the deals that were missed because of Internet connection?"
"How many competitors were there in the trade with the Australian government?"
"What is the ratio between missed calls and volume lost?"
"Which deals were lost by the marketeer Mary Adams?"
```

When the table changed, we were hoping to keep as much as we could of the implemented work. Although the domain of each table is different, we managed to replicate the process that had been done for the first table. The logic for the implementation was the same: characterize the domain, create categories, gather synonyms and create rules that would cover the range of questions that could be asked by the user.

It is important to mention that when using a rule-based approach, each implementation is specific to the domain. For that matter, when changing to the new table, the new domain had to be analyzed, and the new possible questions studied. One can also see that the *categories* defined for the *Rollover Opportunities* table could not be obtained from the work that was done for this table, since the questions have different structures and use different words. Therefore, the learning here was how crucial the table and its domain is, and that one rule-based system cannot be plugged into another.

4.6 Summary

Our evaluation tested the full pipeline: NLP processing, question analysis, and both rule-based approaches, sys1 and sys2 .

We noticed that every detail is relevant in a rule-based approach, and that small alterations to the *user question*, and mentioned *named entities*, change the accuracy of the system completely. We have seen how reformulating the question can lead to an answer, and also saw how the users can improve their interaction with the system by learning about its limitations. We have also seen how improving our *grammar rules* and increasing our list of *synonyms* will increase the number of questions the NLIDB can answer.

Since our domain changed, we also had the opportunity of studying how rule-based approaches depend on the domain they are being implemented in.

5

Contributions, Conclusions and Future Work

Contents

5.1 Contributions

This thesis contributed with an NLIDB to be used in the GCA group.

The GCA group now has access to the research and implementation of this NLIDB to decide if a rule-based approach is useful for their many dashboards and tables. Since this project implemented two approaches, sys1 and sys2, they can also see how one approach compares to the other in terms of implementation details and results.

5.2 Conclusions

In this project we have studied different NLIDBs and NLP systems. We have studied a financing domain, and planned how to implement a NLIDB that would create SQL queries for a table.

We have implemented a pipeline, starting with several NLP methods, so that the user's question can be processed. After that, our question is sent to two different rule-based grammars: a semantic grammar and a grammar with a *learning* perspective. This allowed us to see the differences in the implementation of these two systems, and how using date expressions or different grammar systems requires different questions to be submitted.

With our development tests, we noticed that systems that use NL are very sensitive, and that extra attention needs to be given to every detail. With our user tests, we saw the importance of the user understanding the system, since the user can then adapt and consequently, the interaction with the system will be better.

5.3 Future Work

In this section we will focus on some aspects that can enhance the work so far presented in this document.

The pool of questions that the system is capable of answering can be augmented, since there are further combinations that can be considered. There are multiple ***Attributes*** and ***Measures***, and their combinations of three or four, may be useful for the user. We would need to have a natural language understanding component prepared to deal with those *conjunctions*. However, we should keep in mind that *conjunctions* are a difficulty in all NLP systems because of their multiple possible meanings.

Ellipses are another characteristic of natural language that can be approached. *Ellipses'* difficulty come from the system's need to understand what word the ellipsis refers to. It would however, allow the user to ask multiple questions referring to the same entry of the table without having to re-ask for things the system has already answered.

We could also change the learning mechanism, and test other mechanisms. The SEMPRE toolkit

itself has a feedback learning mechanism that was not used. As mentioned in Section 2.8, the mechanism consists of the user informing the system of the rule that is correct, and should have been used. This is helpful in the case of multiple rules firing for the same question. The system will accept the user changes as the correct answer, and increases the probability of that rule, that will then be active in the next interaction.

If we were able to acquire an large quantities of user data based on the uses of the NLIDB, we could also obtain patters from it. There are big data methods that allow an analysis of the questions asked and how they can be used to improve the system.

Bibliography

[1] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases–an introduction," *Natural language engineering*, vol. 1, no. 1, pp. 29–81, 1995.

[2] D. H. Warren and F. C. Pereira, "An efficient easily adaptable system for interpreting natural language queries," *Computational Linguistics*, vol. 8, no. 3-4, pp. 110–122, 1982.

[3] B. H. Thompson and F. B. Thompson, "Introducing ask, a simple knowledgeable system," in *Proceedings of the first conference on Applied natural language processing*. Association for Computational Linguistics, 1983, pp. 17–24.

[4] M. L. Mauldin, "Chatterbots, tinymuds, and the turing test: Entering the loebner prize competition," in *AAAI*, vol. 94, 1994, pp. 16–21.

[5] J. Weizenbaum, "Eliza — a computer program for the study of natural language communication between man and machine," *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, 1966.

[6] C. Thompson, "Approximating life," *New York Times Magazine*, vol. Late Edition, no. July, 7 2002, pp. 30–33, 2002.

[7] R. S. Wallace, "The anatomy of alice," in *Parsing the Turing Test*. Springer, 2009, pp. 181–210.

[8] P. Norvig, *Paradigms of artificial intelligence programming: Case studies in Common LISP*. Morgan Kaufmann, 1992.

[9] B. A. Shawar and E. Atwell, "Chatbots: are they really useful?" in *Ldv Forum*, vol. 22, 2007, pp. 29–49.

[10] V. Han, "Are chatbots the future of training ?" in *Learning Technologies*, vol. 71, 2017, pp. 42–46.

[11] J. Hand, *ChatOps*. O'Reilly Media, Inc, 2016.

[12] W. Tunstall-Pedoe, "True knowledge: Open-domain question answering using structured knowledge and inference," *AI Magazine*, vol. 31, no. 3, pp. 80–92, 2010.

[13] A. B. Abacha and P. Zweigenbaum, "Means: A medical question-answering system combining nlp techniques and semantic web technologies," *Information processing & management*, vol. 51, no. 5, pp. 570–594, 2015.

[14] X. Li and D. Roth, "Learning question classifiers: the role of semantic information," *Natural Language Engineering*, vol. 12, no. 3, pp. 229–249, 2006.

[15] J. P. C. G. Silva, "Qa+ml@wikipedia&google," Master's thesis, Instituto Superior Técnico – Universidade de Lisboa, 2009.

[16] O. Williams, "High-performance question classification using semantic features," *Standford University*, pp. 1–7, 2010.

[17] A.-M. Popescu, O. Etzioni, and H. Kautz, "Towards a theory of natural language interfaces to databases," in *Proceedings of the 8th international conference on Intelligent user interfaces*. ACM, 2003, pp. 149–157.

[18] M. d'Aquin and E. Motta, "Watson, more than a semantic web search engine," *Semantic Web*, vol. 2, no. 1, pp. 55–63, 2011.

[19] F. Li and H. Jagadish, "Constructing an interactive natural language interface for relational databases," *Proceedings of the VLDB Endowment*, vol. 8, no. 1, pp. 73–84, 2014.

[20] T.-H. Wen, D. Vandyke, N. Mrksic, M. Gasic, L. M. Rojas-Barahona, P.-H. Su, S. Ultes, and S. Young, "A network-based end-to-end trainable task-oriented dialogue system," *arXiv preprint arXiv:1604.04562*, 2016.

[21] D. Song, F. Schilder, C. Smiley, C. Brew, T. Zielund, H. Bretz, R. Martin, C. Dale, J. Duprey, T. Miller *et al.*, "Tr discover: A natural language interface for querying and analyzing interlinked datasets," *Lecture Notes in Computer Science*, vol. 9367, pp. 21–37, 2015.

[22] D. Deutch, N. Frost, and A. Gilad, "Provenance for natural language queries," *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 577–588, 2017.

[23] W. Wahlster, H. Marburger, A. Jameson, and S. Busemann, "Over-answering yes-no questions," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence. Karlsruhe, West Germany*, 1983, pp. 643–646.

[24] Y. Cao, F. Liu, P. Simpson, L. Antieau, A. Bennett, J. J. Cimino, J. Ely, and H. Yu, "Askhermes: An online question answering system for complex clinical questions," *Journal of biomedical informatics*, vol. 44, no. 2, pp. 277–288, 2011.

[25] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer, "Learning a neural semantic parser from user feedback," *arXiv preprint arXiv:1704.08760*, pp. 963–973, 2017.

[26] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager *et al.*, "Building watson: An overview of the deepqa project," *AI magazine*, vol. 31, no. 3, pp. 59–79, 2010.

[27] A. Lally and P. Fodor, "Natural language processing with prolog in the ibm watson system," *The Association for Logic Programming (ALP) Newsletter*, pp. 1–4, 2011.

[28] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates, "Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability," in *Proceedings of the 20th international conference on Computational Linguistics*. Association for Computational Linguistics, 2004, p. 141.

[29] D. Kuepper, M. Storbel, and D. Roesner, "Nauda: A cooperative natural language interface to relational databases," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 529–533, 1993.

[30] D. Song, F. Schilder, C. Smiley, and C. Brew, "Natural language question answering and analytics for diverse and interlinked datasets," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, 2015, pp. 101–105.

[31] R. High, "The era of cognitive systems: An inside look at ibm watson and how it works," *International Business Machines Corporation, Redbooks*, pp. 1–14, 2012.

[32] J. Silva, L. Coheur, A. C. Mendes, and A. Wichert, "From symbolic to sub-symbolic information in question classification," *Artificial Intelligence Review*, vol. 35, no. 2, pp. 137–154, 2011.

[33] A. C. Mendes, L. Coheur, J. Silva, and H. Rodrigues, "Just.ask – a multi-pronged approach to question answering," *International Journal on Artificial Intelligence Tools*, vol. 22, no. 01, p. 1250036, 2013.

[34] C. L. Bates, P. R. Day, and J. M. Santosuosso, "Ordering of database search results based on user feedback," Mar. 29 2005, uS Patent 6,873,982.

[35] J. Berant, A. Chou, R. Frostig, and P. Liang, "Semantic parsing on freebase from question-answer pairs," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2013.

A

Table Fields

A.1 Measure List for "Rollover Opportunities" database

Volume

TotalTrades

StdDevRolloverDays

Spot Price Strike

RolloverRatio

Rank

NetVolumeRatio

NetVolume

NearLegVolume

LatestRolloverDays

EV

CC Total

CC NonRisk

CC AtRisk

AverageRolloverDays

MaxAbsNetVolume

A.2 Attribute List for "Rollover Opportunities" database

TradeStatus

Tenor

Product Group

Product

Platform

NewTrade

NetClientPosition

NearTenorDays

MaxFutureDate

Marketer

LocalBlotter

Leg

FromDays

FarTenorDays

Expiry Date

Deal ID

Deal Date

Date Today

CurrencyPairGroup

Currency Pair

CRDSCode

Client Deal Side

Client

Broker_FXT

NDF Fixing Date

B

Dashboard definitions

Rollover: A rollover is the process of extending the settlement date of an open position. https://www.investopedia.com/ask/answers/06/rollover.asp.

Outright: An outright forward is a forward currency contract that locks in an exchange rate for a specific delivery date and a specific amount. https://www.investopedia.com/terms/o/outright-forward.asp.

Swap: A foreign currency swap is an agreement to exchange currency between two foreign parties. https://www.investopedia.com/terms/f/foreign-currency-swaps.asp.

NearDate, FarDate: 2 legs of the swaps. A FX Swap has two legs or stages (a near leg date and a far leg date). On the near leg date, you swap one currency for another at an agreed spot foreign exchange rate and agree to swap the same currencies back again on a future date (far leg date) at a forward foreign exchange rate.

Hedge: A hedge is an investment to reduce the risk of adverse price movements in an asset. https://www.investopedia.com/terms/h/hedge.asp.

CRDS Code: Counterparty Code – account for the client where he is allowed to trade. Code that represents the entity of a client. There are 2 types of CRDS Codes:

–**Legal Entity:** 7 characters: first 4 refresent the client and the other 3 the location; A legal Entity is a legally responsible counterparty that holds its own capital and with whom BNP Paribas does business.

–**Non Legal Entity:** 12 characters: first 7 represent the legal entity code and the other 5 the non legal entity code; A non legal entity may be defined as a hierarchical splitting of a legal entity (department, agency) that makes business under its own name but is not legally independent.

Currency Pair: Value of the base and quote currency (ex: EUR/USD). https://www.investopedia.com/terms/c/currencypair.asp.

Client Direction: Direction at the client side: Buy or Sell.

Client Contribution (CC): Client Contribution measures the expected revenues of a client trade at inception It is the amount of money attached to a deal attributed to Marketers to measure the contribution of the client to the revenue of the bank The total CC is made up of two components:

–At Risk (AR)
–Added Value/Non Risk amount (AV)

Client Contribution cannot be negative in normal circumstances unless in cases of a Commercial Gesture or an Operational Risk Incident

Tenor: is the amount of time left for the repayment of a loan or contract or the initial term length of a loan. https://www.investopedia.com/terms/t/tenor.asp.

NetVolume: Volume that takes into account the direction of the trade: Buy (positive value) or Sell (negative value) if using BNP view. Opposite signs if it is the client view.

Economic Value (EV): Attributed to the FX Business Unit. An estimate of client revenues generated by a trade (currently only available for Flow FX products). Also known as the actual 'post-trade' revenue EV defaults to CC.

C

sys1 Grammar

```
import ply.lex as lex
import ply.yacc as yacc
tokens = ( 'MEASURE', 'ATTRIBUTE', 'EXCLUDE', 'VALUE', 'COMPARE', 'DATE')

# Tokens

t_MEASURE = r'volume | totaltrades | stddevrolloverdays | spot \s price \s strike | rolloverratio | rank | netvolumeratio | netvolume'
            r'nearlegvolume | latestrolloverdays | ev | cc \s total | cc\s nonrisk | cc \s atrisk | averagerolloverdays | maxabsnetvolume'

t_ATTRIBUTE = r'tradestatus | tenor | product \s group | product | platform | newtrade | netclientposition | neartenordays | '
            r'maxfuturedate | marketer | localblotter | leg | fromdays | fartenordays | expiry \s date | deal \s id | deal \s date'
            r'| currencypairgroup | currency \spair | crdscode | client \s deal \s side | client'
            r'| broker_fxt | ndf | fixing \s date'

t_VALUE = r'([a-zA-Z]3[/][a-zA-Z]3) | [a-zA-Z_0-9]+ '

t_DATE = r'([0-9]4[-][0-9]2[-][0-9]2) | ([0-9]9)'

def t_EXCLUDE(t):
r'exclude'
return t

def t_COMPARE(t):
r'highest'
return t

# Ignored characters
t_ignore = " \t"

def t_newline(t):
r'\n+'
```

```
t.lexer.lineno += t.value.count("\n")

    def t_error(t):
print("Illegal character '%s'" % t.value[0])
t.lexer.skip(1)

    # Build the lexer
lex.lex()

    # Dictionary of names (for storing variables)
names = {}

    # Rules

    def p_statement_date(p):
'''statement : ATTRIBUTE DATE'''
        p[0] = "".join(("SELECT * FROM table WHERE ", "[", p[1], "]", " = ", " ' ", p[2], " ' ",))

    def p_statement_attribute_date(p):
'''statement : ATTRIBUTE ATTRIBUTE DATE
                | MEASURE ATTRIBUTE DATE'''
        p[0] = "".join(("SELECT ", "[", p[1], "]", " FROM table WHERE ", "[", p[2], "]", " = ", " ' ",
p[3], " ' "))

    def p_statement_date_date(p):
'''statement : ATTRIBUTE DATE DATE'''
        p[0] = "".join( ("SELECT * FROM table WHERE ", "[", p[1], "]", " ≥ ",
" ' ", p[2], " ' ", " AND ", "[", p[1], "]", " ≤ ", " ' ", p[3], " ' "))

    def p_statement_attribute_date_date(p):
'''statement : ATTRIBUTE ATTRIBUTE DATE DATE
                | MEASURE ATTRIBUTE DATE DATE'''
        p[0] = "".join( ("SELECT ", "[", p[1], "]", " FROM table WHERE ", "[", p[2], "]", " ≥", " ' ",
p[3], " ' ", " AND ", "[", p[2], "]", " ≤ ", " ' ", p[4], " ' "))
```

```
    def p_statement_single(p):
"" statement : MEASURE VALUE
             | ATTRIBUTE VALUE ""
        p[0] = "".join(("SELECT * FROM table WHERE ", "[", p[1], "]", " = ", " ' ", p[2], " ' "))

    def p_statement_single_second(p):
"" statement : VALUE MEASURE
             | VALUE ATTRIBUTE ""
        p[0] = "".join(("SELECT * FROM table WHERE ", "[", p[2], "]", " = ", " ' ", p[1], " ' "))

    def p_statement_exclude(p):
"" statement : MEASURE VALUE EXCLUDE
             | ATTRIBUTE VALUE EXCLUDE ""
        p[0] = "".join(("SELECT * FROM table WHERE ", "[", p[1], "]", " != ", " ' ", p[2], " ' "))

    def p_statement_exclude_middle(p):
"" statement : MEASURE EXCLUDE VALUE
             | ATTRIBUTE EXCLUDE VALUE ""
        p[0] = "".join(("SELECT * FROM table WHERE ", "[", p[1], "]", " != ", " ' ", p[3], " ' "))

    def p_statement_exclude_first(p):
"" statement : EXCLUDE MEASURE VALUE
             | EXCLUDE ATTRIBUTE VALUE ""
        p[0] = "".join(("SELECT * FROM table WHERE ", "[", p[2], "]", " != ", " ' ", p[3], " ' "))

    def p_statement_exclude_large(p):
"" statement : MEASURE EXCLUDE ATTRIBUTE VALUE
             | ATTRIBUTE EXCLUDE MEASURE VALUE
             | ATTRIBUTE EXCLUDE ATTRIBUTE VALUE
             | MEASURE EXCLUDE MEASURE VALUE ""
        p[0] = "".join( ("SELECT ", "[", p[1], "]", " FROM table WHERE ", "[", p[3], "]", " != ", " ' ",
p[4], " ' "))

    def p_statement_multiple_value_second(p):
"" statement : ATTRIBUTE VALUE MEASURE
```

```
                | MEASURE VALUE ATTRIBUTE
                | ATTRIBUTE VALUE ATTRIBUTE "'
        p[0] = "".join( ("SELECT ", "[", p[3], "]", " FROM table WHERE ", "[", p[1], "]", " = ", " ' ",
p[2], " ' "))

    def p_statement_multiple_value_third(p):
"' statement : ATTRIBUTE MEASURE VALUE
                | MEASURE ATTRIBUTE VALUE
                | ATTRIBUTE ATTRIBUTE VALUE "'
        p[0] = "".join( ("SELECT ", "[", p[1], "]", " FROM table WHERE ", "[", p[2], "]", " = ", " ' ",
p[3], " ' "))

    def p_statement_multiple_value_four(p):
"' statement : ATTRIBUTE VALUE MEASURE VALUE
                | MEASURE VALUE ATTRIBUTE VALUE
                | ATTRIBUTE VALUE ATTRIBUTE VALUE"'
        p[0] = "".join(("SELECT * FROM table WHERE ", "[", p[1], "]", " = ", " ' ", p[2], " ' ", " AND
", "[", p[3], "]", " = ", " ' ", p[4], " ' "))

    def p_statement_compare(p):
"' statement : ATTRIBUTE COMPARE MEASURE
                | MEASURE COMPARE ATTRIBUTE
                | ATTRIBUTE COMPARE ATTRIBUTE "'
        p[0] = "".join( ("SELECT TOP 10 ", "[", p[3], "]", ", ", "[", p[1], "]", " FROM table "))

    def p_statement_compare_single(p):
"' statement : COMPARE MEASURE
                | COMPARE ATTRIBUTE "'
        p[0] = "".join( ("SELECT TOP 10 ", "[", p[2], "]", " FROM table "))

    def p_error(p):
print(p)
print("Syntax error at '%s' " % p.value)

    yacc.yacc()
```

D

sys2 Grammar

```
#——— MEASURE RULES
# Basic Measure rules where the measure is defined as a string constant
(rule $Measure (volume) (ConstantFn (string "[Volume]")))
(rule $Measure (totaltrades) (ConstantFn (string "[TotalTrades]")))
(rule $Measure (stddevrollover) (ConstantFn (string "[StdDevRolloverDays]")))
(rule $Measure (spotpricestrike) (ConstantFn (string "[Spot Price Strike]")))
(rule $Measure (rolloverratio) (ConstantFn (string "[RolloverRatio]")))
(rule $Measure (rank) (ConstantFn (string "[Rank]")))
(rule $Measure (netvolumeratio) (ConstantFn (string "[NetVolumeRatio]")))
(rule $Measure (netvolume) (ConstantFn (string "[NetVolume]")))
(rule $Measure (nearlegvolume) (ConstantFn (string "[NearLegVolume]")))
(rule $Measure (rollover) (ConstantFn (string "[LatestRolloverDays]")))
(rule $Measure (ev) (ConstantFn (string "[EV]")))
(rule $Measure (cctotal) (ConstantFn (string "[CC Total]")))
(rule $Measure (ccnonrisk) (ConstantFn (string "[CC NonRisk]")))
(rule $Measure (ccatrisk) (ConstantFn (string "[CC AtRisk]")))
(rule $Measure (avgrollover) (ConstantFn (string "[AverageRolloverDays]")))
(rule $Measure (maxabsnetvolume) (ConstantFn (string "[MaxAbsNetVolume]")))


#——— ATTRIBUTE RULES
# Basic Attribute rules where the attribute is defined as a string constant
(rule $Attribute (tradestatus) (ConstantFn (string "[TradeStatus]")))
(rule $Attribute (tenor) (ConstantFn (string "[Tenor]")))
(rule $Attribute (productgroup) (ConstantFn (string "[Product Group]")))
(rule $Attribute (product) (ConstantFn (string "[Product]")))
(rule $Attribute (platform) (ConstantFn (string "[Platform]")))

(rule $Attribute (newtrade) (ConstantFn (string "[NewTrade]")))
(rule $Attribute (netclientposition) (ConstantFn (string "[NetClientPosition]")))
(rule $Attribute (neartenordays) (ConstantFn (string "[NearTenorDays]")))
(rule $Attribute (maxfuturedate) (ConstantFn (string "[MaxFutureDate]")))
(rule $Attribute (marketer) (ConstantFn (string "[Marketer]")))
(rule $Attribute (localblotter) (ConstantFn (string "[LocalBlotter]")))
(rule $Attribute (leg) (ConstantFn (string "[Leg]")))
(rule $Attribute (fromdays) (ConstantFn (string "[FromDays]")))
```

(rule $Attribute (fartenordays) (ConstantFn (string “[FarTenorDays]”)))
(rule $Attribute (expire) (ConstantFn (string “[Expiry Date]”)))
(rule $Attribute (deal) (ConstantFn (string “[Deal ID]”)))
(rule $Attribute (date) (ConstantFn (string “[Deal Date]”)))
(rule $Attribute (currencypairgroup) (ConstantFn (string “[CurrencyPairGroup]”)))
(rule $Attribute (currency) (ConstantFn (string “[Currency Pair]”)))
(rule $Attribute (crdscode) (ConstantFn (string “[CRDSCode]”)))
(rule $Attribute (clientdeal) (ConstantFn (string “[Client Deal Side]”)))
(rule $Attribute (client) (ConstantFn (string “[Client]”)))
(rule $Attribute (broker) (ConstantFn (string “[Brooker_FXT]”)))
(rule $Attribute (ndfdate) (ConstantFn (string “[NDF Fixing Date]”)))

#——— INDIVIDUAL RULES
Rules that will define key words
(rule $Exclude (exclude) (ConstantFn (string “exclude”)))

#——— COMPARE RULES
This rule allows the client to use the word “highest”
(rule $Compare (highest) (ConstantFn (string “highest”)))

#——— GENERAL RULES

(rule $General ($Attribute $VALUE) (lambda d (lambda t (call + (string “SELECT * ”) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ = ”) (string “ ‘”) (var t) (string “’”)))))

(rule $General ($VALUE $Attribute) (lambda t (lambda d (call + (string “SELECT * ”) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ = ”) (string “ ‘”) (var t) (string “’”)))))

(rule $General ($Measure $VALUE) (lambda m (lambda t (call + (string “SELECT * ”) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ = ”) (string “ ‘”) (var t) (string “’”)))))

(rule $General ($VALUE $Measure) (lambda t (lambda m (call + (string “SELECT * ”) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ = ”) (string “ ‘”) (var t) (string “’”)))))

(rule $General ($Attribute $VALUE $Measure) (lambda d (lambda t (lambda m (call + (string “SE-

LECT ”) (var m) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ = ”) (string “ ‘”) (var t) (string “ ‘”))))))

(rule $General ($Measure $VALUE $Attribute) (lambda m (lambda t (lambda d (call + (string “SELECT ”) (var d) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ = ”) (string “ ‘”) (var t) (string “”))))))

(rule $General ($Attribute $Measure $VALUE) (lambda d (lambda m (lambda t (call + (string “SELECT ”) (var d) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ = ”) (string “ ‘”) (var t) (string “”))))))

(rule $General ($Measure $Attribute $VALUE) (lambda m (lambda d (lambda t (call + (string “SELECT ”) (var m) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ = ”) (string “ ‘”) (var t) (string “”))))))

(rule $General ($Attribute $Compare $Measure) (lambda d (lambda m (lambda x (call + (string “SELECT TOP 10 ”) (var x) (string “ , ”) (var d) (string “ FROM table ”))))))

(rule $General ($Attribute $Compare $Attribute) (lambda d (lambda m (lambda x (call + (string “SELECT TOP 10 ”) (var x) (string “ , ”) (var d) (string “ FROM table ”))))))

(rule $General ($Measure $Compare $Attribute) (lambda m (lambda x (lambda d (call + (string “SELECT TOP 10 ”) (var d) (string “ , ”) (var m) (string “ FROM table ”))))))

(rule $General ($Compare $Attribute) (lambda x (lambda a (call + (string “SELECT TOP 10 ”) (var a) (string “ FROM table ”)))))

(rule $General ($Compare $Measure) (lambda x (lambda a (call + (string “SELECT TOP 10 ”) (var a) (string “ FROM table ”)))))

(rule $General ($Attribute $VALUE $Attribute) (lambda d (lambda t (lambda c (call + (string “SELECT ”) (var c) (string “ FROM table ”)
(string “WHERE ”) (var d) (string “ = ”) (string “ ‘”) (var t) (string “”))))))

(rule $General ($Attribute $Attribute $VALUE) (lambda d (lambda m (lambda t (call + (string

“SELECT “) (var d) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ = ”) (string “ ‘”) (var t) (string “”))))))

(rule $General ($Attribute $Exclude $Measure $VALUE) (lambda d (lambda e (lambda m (lambda t (call + (string “SELECT ”) (var d) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ != ”) (string “ ‘”) (var t) (string “”)))))))

(rule $General ($Measure $Exclude $Attribute $VALUE) (lambda d (lambda e (lambda m (lambda t (call + (string “SELECT ”) (var d) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ != ”) (string “ ‘”) (var t) (string “”)))))))

(rule $General ($Measure $Exclude $Measure $VALUE) (lambda d (lambda e (lambda m (lambda t (call + (string “SELECT ”) (var d) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ != ”) (string “ ‘”) (var t) (string “”)))))))

(rule $General ($Attribute $Exclude $Attribute $VALUE) (lambda d (lambda e (lambda m (lambda t (call + (string “SELECT ”) (var d) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ != ”) (string “ ‘”) (var t) (string “”)))))))

(rule $General ($Measure $Exclude $VALUE) (lambda m (lambda e (lambda t (call + (string “SELECT *”) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ != ”) (string “ ‘”) (var t) (string “”))))))

(rule $General ($Measure $VALUE $Exclude) (lambda m (lambda t (lambda e (call + (string “SELECT *”) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ != ”) (string “ ‘”) (var t) (string “”))))))

(rule $General ($Attribute $Exclude $VALUE) (lambda d (lambda e (lambda t (call + (string “SELECT *”) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ != ”) (string “ ‘”) (var t) (string “”))))))

(rule $General ($Attribute $VALUE $Exclude) (lambda d (lambda t (lambda e (call + (string “SELECT *”) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ != ”) (string “ ‘”) (var t) (string “”))))))

(rule $General ($Exclude $Measure $VALUE) (lambda e (lambda m (lambda t (call + (string “SELECT *”) (string “ FROM table ”) (string “WHERE ”) (var m) (string “ != ”) (string “ ‘”) (var t) (string “’”))))))

(rule $General ($Exclude $Attribute $VALUE) (lambda e (lambda d (lambda t (call + (string “SELECT *”) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ != ”) (string “ ‘”) (var t) (string “’”))))))

(rule $General ($Attribute $VALUE $Attribute $VALUE) (lambda c (lambda a (lambda d (lambda b (call + (string “SELECT * ”) (string “ FROM table ”) (string “WHERE ”) (var c) (string “ = ”) (string “ ‘”) (var a) (string “’”) (string “ AND ”) (var d) (string “ = ”) (string “ ‘”) (var b) (string “’”)))))))

(rule $General ($Measure $VALUE $Attribute $VALUE) (lambda c (lambda a (lambda d (lambda b (call + (string “SELECT * ”) (string “ FROM table ”) (string “WHERE ”) (var c) (string “ = ”) (string “ ‘”) (var a) (string “’”) (string “ AND ”) (var d) (string “ = ”) (string “ ‘”) (var b) (string “’”)))))))

(rule $General ($Attribute $VALUE $Measure $VALUE) (lambda c (lambda a (lambda d (lambda b (call + (string “SELECT * ”) (string “ FROM table ”) (string “WHERE ”) (var c) (string “ = ”) (string “ ‘”) (var a) (string “’”) (string “ AND ”) (var d) (string “ = ”) (string “ ‘”) (var b) (string “’”)))))))

(rule $General ($Attribute $Date) (lambda d (lambda e (call + (string “SELECT * ”) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ = ”) (string “ ‘”) (var e) (string “’”)))))

(rule $General ($Attribute $Date $Date) (lambda d (lambda e (lambda x (call + (string “SELECT * ”) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ ≥ ”) (var e) (string “ AND ”) (var d) (string “ ≤ ”) (var x))))))

(rule $General ($Attribute $Attribute $Date) (lambda d (lambda x (lambda e (call + (string “SELECT ”) (var d) (string “ FROM table ”) (string “WHERE ”) (var x) (string “ = ”) (var e))))))

(rule $General ($Measure $Attribute $Date) (lambda m (lambda x (lambda e (call + (string “SELECT ”) (var m) (string “ FROM table ”) (string “WHERE ”) (var x) (string “ = ”) (var e))))))

(rule $General ($Attribute $Attribute $Date $Date) (lambda x (lambda d (lambda e (lambda y (call + (string “SELECT ”) (var x) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ ≥ ”) (var

e) (string “ AND ”) (var d) (string “ ≤ ”) (var y)))))))

(rule $General ($Measure $Attribute $Date $Date) (lambda m (lambda d (lambda e (lambda y (call + (string “SELECT ”) (var m) (string “ FROM table ”) (string “WHERE ”) (var d) (string “ ≥ ”) (var e) (string “ AND ”) (var d) (string “ ≤ ”) (var y)))))))

#——— ROOT RULE
(rule $ROOT ($General) (IdentityFn))

#——— DATE RULES
Rules on bottom will be added by the system in runtime

E

sys2 Example File

(example (example (utterance “client x volume”) (targetValue (string “SELECT Volume FROM table WHERE Client = x”)))

(example (utterance “client rolloverratio x”) (targetValue (string “SELECT Client FROM table WHERE RolloverRatio = x”)))

(example (utterance “deal currency eur/usd”) (targetValue (string “SELECT Deal ID FROM table WHERE Currency Pair = EUR/USD”)))

(example (utterance “client highest volume”) (targetValue (string “SELECT TOP 10 Client, Volume FROM table”)))

(example (utterance “deal x expire”) (targetValue (string “SELECT Expiry Date FROM table WHERE Deal = x”)))

(example (utterance “client x expire”) (targetValue (string “SELECT Expiry Date FROM table WHERE Client = x”)))

(example (utterance “client x deal id”) (targetValue (string “SELECT Deal ID FROM table WHERE Client = x”)))

(example (utterance “deal x client name”) (targetValue (string “SELECT Client FROM table WHERE Deal ID = x”)))

(example (utterance “deal x volume”) (targetValue (string “SELECT Volume FROM table WHERE Deal ID = x”)))

(example (utterance “client currency eur/usd”) (targetValue (string “SELECT Client FROM table WHERE Currency Pair = EUR/USD”)))

(example (utterance “deal highest cc total”) (targetValue (string “SELECT TOP 10 Deal ID, CC Total FROM table”)))

(example (utterance “deal platform x”) (targetValue (string “SELECT Deal ID FROM table WHERE Platform = x”)))

(example (utterance “platform excluding x”) (targetValue (string “SELECT * FROM table WHERE Platform != x”)))

(example (utterance “client platform x”) (targetValue (string “SELECT Client FROM table WHERE Platform = x”)))

(example (utterance “client exclude platform x”) (targetValue (string “SELECT Client FROM table WHERE Platform != x”)))

(example (utterance “deal trade status new”) (targetValue (string “SELECT Deal ID FROM table WHERE TradeStatus = new”)))

(example (utterance “client trade status amended”) (targetValue (string “SELECT Client FROM table WHERE TradeStatus = amended”)))

(example (utterance “deal near leg”) (targetValue (string “SELECT Deal ID FROM table WHERE Leg = near”)))

(example (utterance “client far leg”) (targetValue (string “SELECT Client FROM table WHERE Leg = far”)))

(example (utterance “deal client deal sell”) (targetValue (string “SELECT Deal ID FROM table WHERE Client Deal Side = sell”)))

(example (utterance “client client deal buy”) (targetValue (string “SELECT Client FROM table WHERE Client Deal Side = buy”)))

(example (utterance “totaltrades ghjk tenor”) (targetValue (string “SELECT Tenor FROM table WHERE TotalTrades = ghjk”)))

(example (utterance “deal uio expire”) (targetValue (string “SELECT Expiry Date FROM table WHERE Deal ID = uio”)))

(example (utterance “rank client joe”) (targetValue (string “SELECT Rank FROM table WHERE Client = joe”)))