A Virtual Assistant for Web Dashboards

Analytics Bot

Anisa Shahidian
76378
Instituto Superior Técnico
Universidade de Lisboa

Abstract

This work 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 work 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.

1. Introduction

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 work was done in partnership with BNP Paribas and we aimed to simplify this search by creating a Natural Language Interface to Database (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.

1.1 Goal

The goal of this work 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 and 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 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.

1.2 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.

1.3 Document Structure

The remainder of this document is organized as follows:

- Section 2 presents work related to this project.
- Section 3 details the architecture and the implementation of the NLIDB.
- Section 4 evaluates the implementation.
- Section 5 concludes the work.
- Section 6 mentions future work on this topic.

[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.

2. Related Work

This section details the research for this project. We start by describing the historical background of NLIDB's in Section 2.1. We will then look at the main components of NLIDBs in Section 2.2, while also detailing systems that were relevant to our implementation.

2.1 Historical Background

Several systems have tried to provide Natural Language (NL) interpretation over the past decades [2]. 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.

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 2.2.

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* [2] 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 [2] 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 [2] in Listing 1.

Listing 1. Example of a Syntax based grammar. Adapted from [2].

```
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 to complement the grammar present in Listing 1.

Listing 2. Example of a Semantic based grammar. Adapted from [2].

```
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 [13] system), and were meant to maintain a conversation with the user.

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 [8, 15] 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[3], a search engine released in 2009, provides query answers by computing them from external data instead of displaying a list of documents. Even Google[4], has recently been computing the answers to several types of query, particularly mathematical calculations and biography searches.

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[5] and Discord[6], already allow this sort of bot integration [9], as does Google's Allo[7].

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.

2.2 Main Components

Figure 1 shows an overview of the most common Natural Language Interface (NLI)[8] components. The systems have the five major components mentioned in our Goals in Section 1.1: *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

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

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

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

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

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

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

[8] 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.

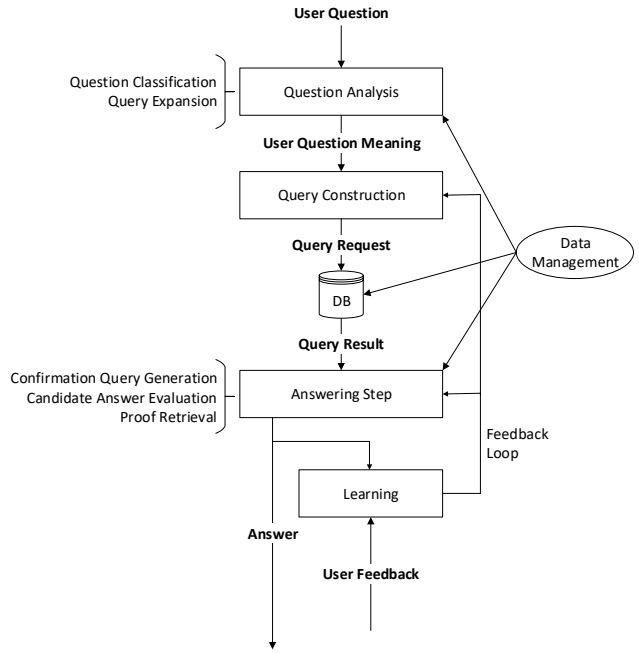

Figure 1. General overview of a NLIDB system.

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 is best. This structuring and storing can be in database, text files, or other means [1, 12, 18].

- **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 [16, 21]. 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*[9].

    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 [14, 16]. 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 [10] [5, 11], while others create a system knowledge base that contain some synonyms or expressions accepted [18].

  It should be noted that in Figure 1, 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 [2], 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 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 [14, 17]. 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)[11] or SPARQL[12] 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 [1], while others consider only one answer at a time, and

[9] A headword can also be called *lemma* and it is the dictionary form, or keyword, for a given meaning or sentence.

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

[11] SQL is a domain-specific language used to manage data stored in databases.

[12] SPARQL is a language to query databases by creating a semantic query.

provide proof with each answer [6, 19]. 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 [4].

- **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 [7, 10].

  Here, we will mention SEMPRE[13][3]. It is a toolkit that allows the creation of semantic parsers, 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) and Part of Speech (POS)[14] 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.

3. Implementation

Based on the work of the previous section, we made several decisions for this implementation. Firstly, for our *Data Management*, we chose to create named entity and synonym lists. Our *Question Processing* will then remove stop-words[15], and perform named entity recognition 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 2.1, 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 *regular expression rule-based approach*, from hereafter called sys1, and a *rule-based approach using the SEMPRE toolkit* (mentioned before in Section 2.2) *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 section will focus on the process of creating the NLIDB. Section 3.1 details the domain of the system. Section 3.2 contains information about how the system functions, detailing the work done before and after a question is submitted to the system. Section 3.5 explains sys1 while Section 3.6 explains sys2. Please note that all data examples in the next chapters have been modified to ensure BNP Paribas data privacy.

3.1 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. 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 3.1.1; the questions and their type will be explained in Section 3.1.2.

3.1.1 Table

The *Rollover Opportunities* table provides information regarding upcoming trade expiry over the next 21 days for Foreign Exchange (FX). The dashboard 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.

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

This division is relevant since we can use these categories to create our grammar and generalize rules.

3.1.2 Questions

The questions that can be asked in this dashboard use both types of fields mentioned above in Section 3.1.1. 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, we can combine the different fields into the grammar sample rules presented in Listing 4.

Listing 3. Example of questions that can be asked by the user. The **Measures** and **Attributes** mentioned are underlined.

"What is *Client* John Doe **Volume**?"
"When will *trade* t567uio9 *expire*?"

"What is *trade* th4ksw1 *currency pair*?"

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

Listing 4. 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 3.1.1 (*Client Name*, *Deal ID*, and *Expiry Date* are ***Attributes***, while *Currency Pair* and

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

[14] Part of Speech creates categories of words with similar grammatical and syntatical properties.

[15] 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.

Volume are ***Measures***), we can observe how the four questions in Listing 3 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 5. 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 5. 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 6.

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 5 using the words "Show me" which is a synonym considered. Section 3.4 will go into further detail.

3.2 Proposal Overview

After researching NLIDBs in Section 2, we had created a general pipeline. Our new pipeline, present in Figure 2, 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.

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), 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 3.4). After the system receives the question, and processes it (**Processed Question**), both approaches implemented are called separately. Section 3.5 will detail sys1, while Section 3.6 will detail sys2, including its learning component.

3.3 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 6, and create

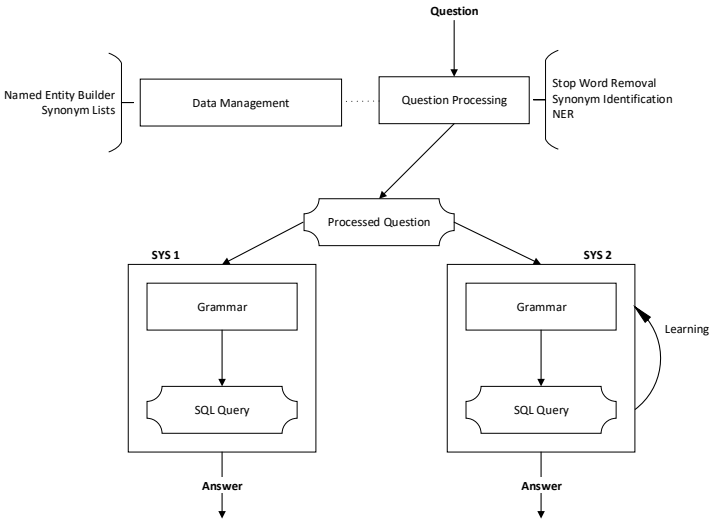

Figure 2. Architecture implemented.

local lists of the table entries to perform *NER*[16]. We have defined[17] the categories based on their field type, and the question structure mentioned in Section 3.1.2. 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 3.4

Listing 6. Query to obtain the values from the table.

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

3.4 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.

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

[16] Named Entity Recognition locates entities in text and classifies them into categories.

[17] 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.

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*.

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 7. 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 7. 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[18] 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 8 has a list of the date expressions that are accepted.

Listing 8. Temporal rules.

```
(next | last) \s([0-9 \s](day | week | year | quarter)

([0-9]4)

(january | february | march | april | may | june | july | august | september
| october | november | december) \s([0-9]4)

((january | february | march | april | may | june | july | august | september
| october | november | december))

yesterday|tomorrow|today|(this year)
```

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 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.

3.5 Sys1

We have created our grammar using a *lexer*[19] and *parser*[20] called PLY[21]. We have included a snippet of the*lexer* in Listing 9, and of the *parser* in Listing 10. 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.

Listing 9. 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
| 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
```

[18] 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.

[19] A lexer is the process of converting characters into tokens with assigned meaning.

[20] A parser takes data (in our case, the tokens created by the parser) and builds a structure while checking for a predefined syntax.

[21] https://github.com/dabeaz/ply. Last accessed August 2018.

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

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 3.1.1. ***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 3.1.2. 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.

As can be seen in Listing 9, ***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[22]. 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 10. 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 10. 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***.

3.6 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 2.2.

The SEMPRE toolkit uses a grammar, that will be explained in Section 3.6.1, which is trained by examples, from which the toolkit provides a score and the probability for each rule the question matches to. Section 3.6.2 will focus on the learning component.

3.6.1 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 grammar implemented contains several types of rules. There are *string declarations*, which declare table entries, ***Attributes***, and ***Measures***, as presented in Listing 11. 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 11. 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[23] such as the ones in Listing 12.

Listing 12. 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 12 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 11 as ***Measure***.

The SEMPRE grammar rules function as part of a tree as can be seen in Figure 3, starting with the rule present in Listing 13.

Listing 13. SEMPRE Root rule.

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

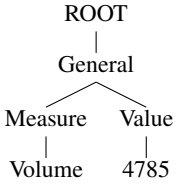

Figure 3. Tree for word matching given the rules provided in Listings 11, 12 and 13

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 by substitution. The SQL query is then returned.

3.6.2 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 2.2. 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 2.2. 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 3.4. 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.

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

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

A sample of this file is present in Listing 14.

Listing 14. Example of SEMPRE learning file.

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

Based on the examples, as mentioned in Section 2.2, 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.

4. Evaluation

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 4.1 will explain how the evaluation was implemented, Section 4.1.1 will detail the corpora used for the tests, and Section 4.1.2 will detail the measure used for the evaluation. The tests performed are described in Section 4.2 and Section 4.3. We conclude this chapter with some considerations about how the domain of NLIDB's influence their implementation in Section 4.4.

4.1 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 2, 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 3.2) 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.

4.1.1 Corpora

To create a corpus that would cover the whole range of questions that could be asked by the user, we created questions to match the existing rules. 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.

4.1.2 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.

4.2 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 15, as well as combinations amongst the ***Measure*** and ***Attribute*** fields, as seen in Listing 17. The rules that match Listing 15's questions are in Listing 16.

Listing 15. 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 16. Rule matches for questions in Listing 15. 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

Listing 17. 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"
```

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.

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 implementation 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 implementation, 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.

4.3 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 18 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 18. 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.

4.4 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 3.1.1. 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 19. 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 3.1.1.

Listing 19. 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.

5. 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.

6. 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 2.2, 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.

References

[1] Abacha, A.B., Zweigenbaum, P.: Means: A medical question-answering system combining nlp techniques and semantic web technologies. Information processing & management 51(5), 570–594 (2015)

[2] Androutsopoulos, I., Ritchie, G.D., Thanisch, P.: Natural language interfaces to databases–an introduction. Natural language engineering 1(1), 29–81 (1995)

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

[4] Cao, Y., Liu, F., Simpson, P., Antieau, L., Bennett, A., Cimino, J.J., Ely, J., Yu, H.: Askhermes: An online question answering system for complex clinical questions. Journal of biomedical informatics 44(2), 277–288 (2011)

[5] d'Aquin, M., Motta, E.: Watson, more than a semantic web search engine. Semantic Web 2(1), 55–63 (2011)

[6] Deutch, D., Frost, N., Gilad, A.: Provenance for natural language queries. Proceedings of the VLDB Endowment 10(5), 577–588 (2017)

[7] Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A.A., Lally, A., Murdock, J.W., Nyberg, E., Prager, J., et al.: Building watson: An overview of the deepqa project. AI magazine 31(3), 59–79 (2010)

[8] Han, V.: Are chatbots the future of training ? In: Learning Technologies. vol. 71, pp. 42–46 (2017)

[9] Hand, J.: ChatOps. O'Reilly Media, Inc (2016)

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

[11] Li, F., Jagadish, H.: Constructing an interactive natural language interface for relational databases. Proceedings of the VLDB Endowment 8(1), 73–84 (2014)

[12] Li, X., Roth, D.: Learning question classifiers: the role of semantic information. Natural Language Engineering 12(3), 229–249 (2006)

[13] Mauldin, M.L.: Chatterbots, tinymuds, and the turing test: Entering the loebner prize competition. In: AAAI. vol. 94, pp. 16–21 (1994)

[14] Popescu, A.M., Etzioni, O., Kautz, H.: Towards a theory of natural language interfaces to databases. In: Proceedings of the 8th international conference on Intelligent user interfaces. pp. 149–157. ACM (2003)

[15] Shawar, B.A., Atwell, E.: Chatbots: are they really useful? In: Ldv Forum. vol. 22, pp. 29–49 (2007)

[16] Silva, J.P.C.G.: QA+ML@Wikipedia&Google. Master's thesis, Instituto Superior Técnico – Universidade de Lisboa (2009)

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

[18] Tunstall-Pedoe, W.: True knowledge: Open-domain question answering using structured knowledge and inference. AI Magazine 31(3), 80–92 (2010)

[19] Wahlster, W., Marburger, H., Jameson, A., Busemann, S.: Over-answering yes-no questions. In: Proceedings of the Eighth International Joint Conference on Artificial Intelligence. Karlsruhe, West Germany. pp. 643–646 (1983)

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

[21] Williams, O.: High-performance question classification using semantic features. Standford University pp. 1–7 (2010)