UNIVERSIDAD DE SEVILLA

MÁSTER UNIVERSITARIO EN LÓGICA, COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

TRABAJO FIN DE MÁSTER

NATURAL LANGUAGE INTERFACES TO RELATIONAL DATABASES

César González Gutiérrez

Dirigido por
José Francisco QUESADA

December 2019

Contents

1 Introduction **1**

2 Background **3**
2.1 Motivation 3
2.2 Historical Review 4
2.2.1 Early Systems 4
2.2.2 Mid-term 6
2.3 State of the Art 7
2.3.1 Machine Learning Techniques 9
2.3.2 Customization, Interactivity and Multimodality 10

3 Approaches to the Problem **13**
3.1 Natural Language Processing Methodologies 13
3.1.1 Symbolic Approach 13
3.1.2 Empirical Approach 14
3.1.3 Hybrid Approach 15
3.2 Architectures 15
3.2.1 Keyword Pattern-Matching Systems 15
3.2.2 Syntax-Based Systems 16
3.2.3 Semantic Grammar Systems 16
3.2.4 Intermediate Representation Systems 17
3.2.5 Dialogue Systems 18
3.3 Challenges 19
3.3.1 Linguistic Issues 20
3.3.2 Interactivity Issues 22
3.3.3 Integration Issues 24
3.3.4 Lessons Learned 25

4 Prototype Design Principles **27**
4.1 Linguistic Approach 27
4.2 Proposed Architecture 27
4.3 Tentative Solutions 29

5 Implementation **31**
5.1 General System Architecture 31
5.2 Interpreter 32
5.2.1 Lexicon 32

5.2.2 Lexical Parser 34
5.2.3 Semantic Parser 36
5.3 Database Query Generator 38
5.3.1 Back-end Communication Channel 38
5.3.2 Database Query Translator 40
5.3.3 Database Communication Layer 42
5.4 Response Generator 43
5.4.1 Response Classifier 43
5.4.2 Answer Generator 43
5.5 Project Organisation and Deployment 44
5.5.1 Skills 45
5.5.2 Micro-services 45

6 Use Cases 47
6.1 Basic Case: Full Pipeline 47
6.2 Linguistic Phenomena 51
6.3 Complex Queries 54

7 Evaluation 57
7.1 Evaluation Framework 57
7.2 Experimental Setup 59
7.3 Results 60

8 Conclusions and future work 61

A Appendix 65
A.1 Containers 65
A.2 Database 68
A.3 Evaluation 70
A.4 Lekta Application REST API 72

Chapter 1

Introduction

The problem to which we devote this work is the creation of a natural language interface to relational databases (NLIDB). This type of databases are still widely used in the industry. But, in order to retrieve information from them, normally a specialist needs to formulate the question in a dedicated query language, such as SQL. The information would be much more accessible if we could retrieve it directly using ordinary language. What an NLIDB system tries to do is, precisely, fill the gap and provide such linguistic interface.

The benefits of such a way of communication with information systems have attracted many researchers in the past, making this one of the oldest natural language processing (NLP) problems to be addressed. Despite the efforts spent trying to give a successful solution, the problem remains open. In this project, we don't intend to solve all the issues associated with the development of this type of system. Instead, we will first focus on the architectural aspects of it. Here, we aim at a general structure that may be flexible enough to be adapted to different scenarios, where specific solutions can be incorporated when needs be. We will also present a functional prototype that will serve as a tester for the proposed architecture.

Attending to the structure of this project, in chapter 2 we will start with a historical review of the different systems that have been developed in the past, in the around 50 years of history of this problem. This will serve the purpose of giving us perspective in time and situate ourselves. Right after, we study the state of the art of this topic by making a literature review of the latests studies.

After presenting the background of this problem, in chapter 3, we will make a survey of the common approaches that have been taken to address this issue. But before going into detail about the architectures the researchers have proposed, we will first look into the different methodologies there exists for natural language processing in general. And we will do so in abstract, attending to the philosophical roots of the techniques in use. Only then we will present a classification of architectures in which the different proposals can be grouped. In this same chapter, we will make a review of the multiple problems these systems must solve in order to perform properly.

At this point, we will have a sufficient understanding of the designs that can be employed to solve this problem more or less successfully. It will be a good time to introduce, in chapter 4, the theoretical principles that our prototype is based on, and how we pretend to face the problems that we have presented before. This will in turn serve to introduce the general structure of our proposal.

In chapter number 5, we get down to work and elaborate on the implementation of the architecture proposed. To give an overall view of the system, first we present a picture of the full pipeline were the user's questions are processed. This general schema depicts the different modules that make up this prototype, and how the data is transformed throughout. Right after, we will go into the details of the different components. At one first level, we will look at the three main functional groups. In a second level of detail, we present the functional components dedicated to more specific tasks, that chained compose the pipeline. In the development of this project, we have taken into account the need for good project organization and ease of deployment. For this reason, we also present the methodology and technologies used to achieve both.

We will gain more clarity about the functioning of the system when we put it to work on some tasks and see how it behaves. In chapter 6, we present a set of use cases that will serve this purpose. We will begin with a basic case, and use it to explain the whole processing pipeline. After that, we will show more elaborate behaviour that becomes apparent when more intricate input is fed into the prototype. We will first highlight linguistic features of the system, that relate more to the linguistic components. Secondly, we will look into more complex queries, that concern the whole system, but more the data transformations that takes place in the back-end components.

The prototype is useful to show some virtues of our approach, by means of displaying desirable behaviours. But it will be very valuable to go a step further and develop an evaluation framework that gives this results a measurable aspect. It is in chapter 7 where we expose our proposal for an evaluation framework for this type of system. During this exposition, we will go thought the implementation of the evaluation framework. Then, the experimental environment we have used to measure the system's capabilities. And, last, the results we have obtained.

We close up this work in chapter 8, summing up with some conclusions. There, we will review the suitability of the approach proposed, the achievements and the aspects that give room for improvement. These refinements will be of service to lead the way to further development in the future.

Chapter 2

Background

2.1 Motivation

A natural language interface to relational databases (NLIDB) is a system that allows a user to formulate questions in some natural language (eg English) about information contained in a relational database management system (RDBMS) and receive the correct answer in response.

The domain of information storage and retrieval has always been related with language, in the sense that we understand the retrieval of information as an act of asking. This metaphor has been always present in the language of specialists in information systems. Expressions like 'query', 'ask the database', 'communicate with the database' are common place in this domain. In RDBMS systems, the metaphor sometimes becomes more explicit in the design of query languages, such as SQL. The syntax of those languages mimic the structure of a natural language question. NLIDB systems try to make the metaphor real, that is, to realise the dream of asking to information systems directly.

There exists many data storage technologies available. The relational model we are talking about was proposed as early as 1970 by E.F. Codd, in his seminal work *A Relational Model of Data for Large Shared Data Banks* [13]. Many other systems have been proposed since then but, even when it seems other models gain track, the relational model remains very popular. It will be interesting to use a well defined model with commercial acceptance as back-end for a NLIDB system.

This type of system could have an important impact in business, making business information available straight away, dropping the costs of information acquisition. Indeed, a survey, quoted in [30], carried out in 1986 on the importance of natural language processing systems, conducted on 33 members of the *Large User Group* professional society, states that: "(1) NLIDBs are the most useful application for organizations among all NLP systems, (2) The five most desirable capacities of NLIDBs are: efficiency, domain independence, pronoun handling, understanding of elliptical entries (i.e., implied words), and processing of sentences with complex nouns, and (3) 50% of the best NLIDBs are those that offer domain independence." Today, these systems aren't less desirable, but the expectations about their performance have relaxed a lot.

2.2 Historical Review

2.2.1 Early Systems

The problem of providing a natural language interface to databases is an old one. The first attempts date back to the late 1960s and early 1970s [14, 5, 7]. This problem has taken a lot of attention from the scientific community, but still we can say that remains open.

The most salient system of the first period was LUNAR [38]. This was a system designed to answer questions about the chemical composition of lunar rocks. The systems of this period were designed for a particular database and not easily portable.

A second period that lasted until late 1970s saw the emergence of new techniques for NLIDBs. During this time, several systems appeared that tried to improve the previous techniques. RENDEZVOUS was one of the firsts to engage the users in dialogues to fulfil their requests. We can say it was the first to use the interaction with the user, that is dialogic features, in order to improve the results obtained. Another interesting system of this time was LADDER [19]. It implemented semantic grammars, "a technique that interleaves syntactic and semantic processing" [5]. It could produce impressive results, but was not easily portable to other databases. Other systems of this period were PLANES and PHILIQA1. We can say this second period was more focused on improving the linguistic and conceptual analysis of queries, but failed at facing the problems related with portability.

In the early eighties of the last century, the focus was shifted to portability issues. For this, researchers started to abandon the semantic grammars, that were the hallmark of the earlier period systems. TEAM [18] tried to ease the burden of configuration, that is, to make this process accessible to database administrators [5]. ASK [34], in the other hand, explored new territory in the improvement of the system by means of user interaction, thus allowing the introduction of new terms and concepts while using it. It also extended the area of application allowing it to connect to other information systems [5, 25]. JANUS [31] had similar connectivity features, acting as an information hub that orchestrated heterogeneous systems in order to fulfil a request formulated in natural language [5, 25].

Other technologies for knowledge representation started to became apparent to approach this problem. These were the golden years of Prolog. CHAT-80 [36], that appeared during this period, was programmed entirely in this programming language [5]. Natural language queries were transformed into logical queries expressed as Prolog statements, then they would be evaluated against the knowledge base of Prolog facts. Another system using this method was MASQUE [6]. These type of systems established the technique of using intermediate representations as a way to dissociate the linguistic expression of a query and its semantic representation, in this case in the form of logical expressions.

By this time, NLIDBs had become a hot topic in the academia. This increasing interest was also followed by companies trying to produce commercially viable products using this technology. Some of these products, available at the time, were INTELLECT from Trinzic, Parlance from BBN, LanguageAccess (IBM's take in this endeavour), also Q&A from Symantec, Natural Language from Natural Language Inc., Loqui from BIM, English Wizard from Linguistic Technology Corp., among others [5].

Despite the numerous attempts to produce systems of this kind that were commercially viable, and the expectations of rapid adoption, after some time the topic started to be relegated to the academic sphere. Companies diverted their attention and efforts to other

Year	Name	Domain	Language	Back-end	Technical approach
1973	LUNAR	Moon rocks	English	DB-specific	grammar
1974	RENDEZVOUS	Airport flights	English	DB-specific	grammar, dialogue
1978	LADDER	US-Navy ships	English	SQL	grammar
1980	CHAT-80	General	English	Prolog	grammar, intermediate representation
1983	TEAM	General	English	Simple Object Database Access	semantic grammar, intermediate representation
1983	ASK	General	English	heterogeneous	intermediate representation
1989	JANUS	General	English	heterogeneous	intermediate representation

Table 2.1: Overview of early NLIDB systems.

products and the database technologies followed a separate path.

This rapid rise of expectations followed by a time of disillusionment is a typical example of what the advisory company Gartner calls a technology hype cycle [16] (see figure 2.1). Broadly speaking, they differentiate five stages in the development of a technology. First comes the *Innovation Trigger*, where a potential technology breakthrough appears. Right after, it follows the *Peak of Inflated Expectations*, where "early publicity produces a number of success stories — often accompanied by scores of failures". This is the time were companies take action and try to deliver the technology, expecting wide acceptance. Then comes the *Trough of Disillusionment* where "interest wanes as experiments and implementations fail to deliver. Producers of the technology shake out or fail". Following this rise and fall, it continues with the *Slope of Enlightenment* where the real benefits of the technology and how it can be integrated are better understood. Finally, it reaches the *Plateau of Productivity* where the viability is neatly defined and where "broad market applicability and relevance are clearly paying off".

In the early 1990s, the topic knew a decrease of interest in the scientific community, which translated into fewer papers published [5]. But it was never fully abandoned. The research followed, divided in different lines of work. Some of those research areas were the adoption of general natural language processing techniques, transforming these systems into reasoning agents or exploring the use of multimodal interfaces.

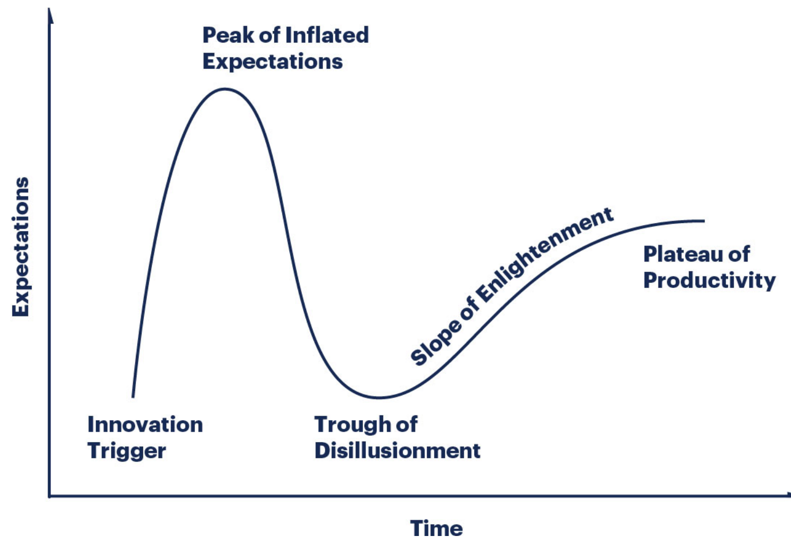

Figure 2.1: Technology Hype Cycle [16].

2.2.2 Mid-term

In the period between late 1990s and the end of 2000s, research on NLIDB systems went through, from a dark period, we can say, until a new resurgence, thanks to the parallel development of new technologies, that opened new lines of research in this topic. The most disruptive technology in this regard was the whole range of statistical modelling and processing, that became popular in this period.

Some developments tried to explore new ways of representing queries as innovative intermediate language representations. One of those is the Conceptual Query Language [26]. The authors of this approach, restrict the types of linguistic expressions the user can produce and the system would accept in the so-called Controlled Natural Language (CNL). This study explores the subset of human language that is easily assimilated to logical representations. From this intermediate representation the system extracts those that are meaningful to the database schema. The technique used to map CNL expressions to database queries are information extraction approaches. The CNL codification of queries also allows the system to make the database queries explainable to some extent.

New information representation technologies were in use at the time, and some systems try to take advantage of them. This is the case of NaLIX (Natural Language Interface for an XML Database), developed in 2006 by the University of Michigan, which is aimed at retrieving data represented in XML format [25]. It produces XQuery expressions, which is a standard query language for XML documents. We can consider this as a syntax based system. Its architecture contains three modules: a parse tree generator, a parse tree validator and a XQuery translation module.

Along these lines, in 2004, the University of Washington developed PRECISE. This development also explores the range of natural language that can be easily transformed into database queries. Here, "the target database is in the form of a relational database using SQL as the query language. It introduces the idea of semantically tractable sentences which are sentences that can be translated to a unique semantic interpretation by analysing some lexicons and semantic constraints" [25]. The system excels in processing the semantically tractable sentences it studies, but it doesn't perform so well with other

types of queries, and particularly struggles with nested structures.

In terms of the study of the similarities between general question answering systems and NLIDB we can cite [33]. The authors consider the QA problem as a superset of the NLIDB problem, that is, as its open domain version. Hence, their approach is inclined to employ the same techniques used in the former in order to answer database queries. In particular, in their prototype “the analysis of the input questions is performed by matching the syntactic parse of the question to a set of fixed templates. This approach is similar to that of much work in QA” [33].

In the line of using statistical methods, we can also mention [10]. In this work, the author divides the system architecture in subsystem that extracts information about the database, and uses that information as a knowledge base. Another component extracts patterns from a corpus of queries in the form of graphs. This last operation is done using statistical pattern extraction. These two sources of knowledge are then merged by another component to generate the final database query.

Another system produced during this period was WASP. Developed by a team at the University of Texas, it focuses in the wider problem of producing a universal representation of sentences as a formal, symbolic representation of its meaning. It uses Prolog as means of expressing its queries. This system learns how to translate sentences from the natural language into this formal query language. For this task, it uses statistical machine translation techniques [25]. Thanks to this, it can be adapted to different domains, assuming the corresponding dataset is provided. Similar to other systems based on statistical methods, the corpus they use need to destroy the information in the database structure first and can’t take advantage of it.

2.3 State of the Art

A literature review of the current developments shows that there exists still an important interest in this area of research. We can broadly classify the recent surveys in two distinct blocks apart from the advances on the pre-existing techniques. In a first block we group the different techniques developed with the help of advances in the field of machine learning. These studies provide with new techniques to substitute existing components in earlier architectures or inspire new combinations to improve the existing ones. Second, we can talk about a second group of systems that are more focused on the pragmatics of querying an information system. In this list we find new ways of interaction with the systems, new modalities of communication, and also new ways to facilitate the customization. The improvements on classic approaches have focussed mainly on finding new ways of representing the knowledge as intermediate languages.

Looking at the commercial aspect of this technology, it appears these systems have fallen somewhat into disuse. The primary type of product that companies tried to commercialize was monolithic systems that, when attached to an existing database or data warehouse, would present users with a linguistic interface. This type of product is clearly in decay. The list of discontinued products bears witness to this fact. LanguageAccess, developed by IBM, English Query from Microsoft and DataTalk, commercialized by Natural Language Inc., among others, all have been discontinued [28].

But, far from being abandoned, this technology now lives within other lines of products, mixed with other linguistic interfaces to information systems, and, many times, goes

			Stop word	Synonym	PoS	Stem / Lemma	NER	Dep. Parser	Const. Parser	SQL	SPARQL	Other
Keyword	**SODA**	*2012*	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗
	NLP-Reduce	*2007*	✗	✓	✗	✓	✗	✗	✗	✓	✗	✓
	Précis	*2008*	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
	QUICK	*2009*	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗
	QUEST	*2013*	?	?	?	?	✗	✗	✗	✓	✗	✗
	SINA	*2015*	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗
	Aqqu	*2015*	?	✓	✓	✓	✓	✗	✗	✗	✓	✗
Pattern	**NLQ/A**	*2017*	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗
	QuestIO	*2008*	✗	?	✓	✓	✗	✗	✗	✗	✓	✗
Parsing	**ATHENA**	*2016*	✗	✓	✓	✓	✓	✓	✗	✓	✗	✗
	Querix	*2006*	✗	✓	✓	?	✗	✗	✓	✗	✓	✗
	FREyA	*2010*	?	✓	✓	?	✗	✗	✓	✗	✓	✗
	BELA	*2012*	✗	✓	✓	?	✗	✗	✓	✗	✓	✗
	USI Answers	*2013*	?	✓	✓	✓	✓	✓	✗	✓	✓	✓
	NaLIX	*2005*	✗	✓	✓	?	✗	✗	✓	✗	✗	✓
	NaLIR	*2014*	✗	✓	✓	?	✗	✗	✓	✓	✗	✗
	BioSmart	*2017*	?	?	✓	?	✗	✗	✓	✓	✗	✓
Grammar	**TR Discover**	*2015*	?	▲	✗	✗	✗	✗	✗	✓	✓	✗
	Ginseng	*2005*	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗
	SQUALL	*2014*	✗	✓	✓	✗	✗	✗	✓	✗	✓	✗
	MEANS	*2015*	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗
	AskNow	*2016*	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗
	SPARKLIS	*2017*	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
	GFMed	*2017*	?	?	?	?	?	?	?	✗	✓	✗

✓, using; ▲, partly using; ✗, not using; ?, not documented

Figure 2.2: Categorization of recent NLIDB systems and the used NLP technologies [2].

unnoticed in commercial applications. This is the case of Google's search engine, which includes some question answering capabilities, or Siri from Amazon, and some others [2].

The tendency seems to be moving from the monolithic database interface and combine this technique with others to improve the results, in particular in analytics and business intelligence. Several companies have developed products in this line, and they have received important financial support [11]. This speaks to the fact that there is still hope to achieve quality products using this technology. Apart from the companies mentioned before, [11] mentions several others. In this list we can include Thoughtspot[1], founded in 2012, Arimo's Narratives [2] (2012), Power BI[3] (2015) by Microsoft or Wolfram Alpha[4] (2009).

To give a general picture of the current NLIDBs, in figure 2.2 the reader can see a table with a summary of recent systems and the use of Natural Language Processing techniques they employ [2]. The techniques mentioned here, will come across when we talk about the prototype architecture later in this document. Also, the major classification used in this table, will be better understood throughout the exposition of types of architectures in section 3.2.

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

[2] https://arimo.com/products/narratives/

[3] https://powerbi.microsoft.com/en-us/

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

2.3.1 Machine Learning Techniques

One of the most relevant factors in the latest studies is the emergence of techniques based on machine learning. In more recent times, it is particularly relevant the advances in neural networks, and the whole topic of Deep Learning.

The optimism (or hype perhaps) about the capacities of Deep Learning techniques in order to model language queries has led to the development of deep learning-based end-to-end systems. These systems "revisit the problem of NLIDBs and recast it as a sequence translation problem" [9]. That is, they try to understand the NLIDB problem as a translation problem, from natural language queries (NLQ) to SQL or other query language.

This kind of systems are faced with a whole new set of problems. In terms of data, "the limiting factor for such systems is the need for a large set of NLQ to SQL pairs for each schema, and consequently some work focuses on the challenge of synthesizing and collecting NLQ-SQL pairs" [8] in order to train them. So far, the most advanced systems of this kind are limited to single-table schemas, due to the difficulty in modelling table relationships. This line of work in NLIDB systems is beyond the scope of this study, for this reason we are not going into detail about the different approaches. But it is interesting to note the three main problems end-to-end systems are facing [21]:

- Domain portability: NLQ-SQL training sets are based on a particular database schema. This makes it very difficult to adapt one model to a new system. Normally the training set has to be generated per schema and this tends to be impossible in practice.

- Explainability: it is interesting for a NLIDB to be able to interact with the user in order to clarify the question or to explain the results given. This requires interpreting some system's intermediate representation. In deep-learning models, intermediate representations are often unexplainable.

- Sources of information: taking into account more sources of information means higher dimensions of input in the dataset. This multiplies the first problem.

There are many ways in which ML techniques can be introduced in a NLIDB system architecture trying to improve the performance of its components. Rethinking the representation of queries can enable the use of statistical methods to map to other structures that can, in turn, produce a final database query. In this line, [17] uses tree-like structures for every query. These trees then define a space of features on which the ML techniques can operate upon. In particular, "a machine learning algorithm is proposed that takes pairs of trees as training input and derives the unknown final SQL query by matching propositional and relational substructures".

ML components need to compile big datasets in order to train their models. At the same time, they need new ways of benchmarking and comparing different approaches. This new class of problems, opens up new fields of study. One of the attempts to face this new problem is [9]. In this paper, the authors build a new dataset based on Stack Exchange Data Explorer website for other ML-based system to train from. They also propose the construction of a neural network-based system. In their approach they understand the NLIDB problem as a sequence translation problem.

These systems can take advantage of several data sources (database schema, natural language question corpus, database query corpus). There were other resources already available that were perhaps underutilized. One of such sources is database logs. For instance, in [21], the author uses database logs as "a representative sample of the distribution of queries". Because the experience shows that the NLQs used fall into a limited number of groups, these templates can be weighted by popularity using logs. The approach to answer a query can now be finding the appropriate template to it. Another interesting characteristic of this approach is the merger of the machine learning teaching machinery in the system's flow.

The authors in [8] also use these logs to extract relevant patterns that can help the performance of NLIDBs. They distinguish two domains were logs can help in "bridging the semantic gap": keyword mapping and join path inference. Keyword mapping refers to the mapping of linguistic entities with database elements (relations, attributes, values). This task is challenging mainly due to the ambiguity of natural language. The second problem consist of inferring the relations between tables. This is necessary in order to connect two database elements that aren't in the same table but are related through some join path. This problem is difficult because the database structure is normally unknown to the users of the system and it must be inferred. To show the performance improvements that NLIDBs can achieve by extracting SQL query log information, they have developed TEMPLAR. In this prototype they use information extraction techniques on the mentioned logs to improve keyword mapping.

2.3.2 Customization, Interactivity and Multimodality

One of the pragmatic problems of this type of system is the cost of configuration for adaptation to new domains. An interesting study in this line is [28]. In this paper we find the history of NLIDBs under the light of this problem. The authors make an exhaustive study of previous systems in terms of customization and evaluation. In order to compare with other system's ability to adapt to new domains, they have developed their own NLIDB with an innovative customization mechanism. This system is based in what they call a Semantic Information Dictionary, that contains all the information that the system then uses to produce the response. The customization process can thus focus on a single element in the architecture.

With the development of the AskMe system [23], the proponents confront this problem directly. Their system "focuses on lowering the costs of technology adoption, portability and users' learning costs" by means of "creating a fully auto-reconfigurable or 'generic interactive'" system. The architecture they propose, that could cope with this level of adaptation, is the "dynamic generation of the lexer, syntactic and semantic parsers".

The study of the use of the system by the users can also provide useful information for improving its performance. Looking at the context of the queries the users enter, [3] can distinguish three different domains of interaction. The authors then are capable of distinguish, in user input, what they call Linear Disjoint interactions, which are the most tractable type of sentence, in order to produce a correct database query response.

The most relevant innovation in recent years in terms of interactivity with information systems, has been the invention of the Internet. In terms of NLIDB interaction modes, this must have been an element to consider. And, precisely, there are architectures that

try to integrate natural language interfaces with web interfaces. This is the case of [4]. As the authors note, web interfaces are a good media to present questioning forms and show table-like query results. The architecture is a template-based processing pipeline. But, the front-end has been substituted by a web page.

Also, focused on interactivity we can mention NaLIR [22]. It is an evolution of NaLIX, explained before in section 2.2.2. Instead of producing XQuery queries from natural language, this new version generates SQL queries. In this system, the authors provide an interactive environment where, with “a carefully limited interaction with the user”, they are able answer complex user questions. At the same time, the system can provide a more interactive experience. The modelling of the queries (a tree structure) allows it to explain the results (or partial results) obtained during the dialogue. This is an interesting feature both in terms of interactivity and for the sake of explainable artificial intelligence, a feature that is becoming more demanding as the use of AI technology spreads and its impact in society widens.

Chapter 3

Approaches to the Problem

3.1 Natural Language Processing Methodologies

The architecture of an NLIDB system can take different shapes depending, first, on the Natural Language Processing approach the designer chooses. Also, depending on the characteristics the system is expected to support, such as interactivity, portability, multi-modality, etc, it will have other requirements. The NLP approach gives us a first broad classification of NLIDB systems.

Depending on the semantics theory we consider in order to model natural language, we will either have compositional semantics or distributional semantics. This distinction is in the core of any approach to Natural Language Processing systems, NLIDBs included. The first school lead us to a symbolic approach, whereas with the second we enter the territory of empirical approaches. These two semantics theories do not necessarily exclude each other. In practical terms we can consider them as complementary. Between the two, there is room for multiple interesting combinations.

3.1.1 Symbolic Approach

From the Wikipedia entry for *Principle of Compositionality* we read: "the principle of compositionality is the principle that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them" [1]. This precept points to certain recursive structure in the core of linguistic expressions that determines its meaning. If we use symbols to represent the elements present in these structures, we have the base for a symbolic approach to Natural Language Processing.

This theory of language can be expressed in a formalism. This is precisely what Noam Chomsky did for language syntax in his seminal work *Syntactic Structures* [12] in 1957. Here we find a formalism were, thanks to a set of symbols and generative rules, we can determine whether a sentence is grammatically correct or not. This is what we know as formal grammar.

We can follow the same approach in order to understand the meaning of a sentence, or a query, for that matter. In an NLIDB system this means processing the natural language query (NLQ) using some rule system that can extract information from its structure. We

[1] https://en.wikipedia.org/wiki/Principle_of_compositionality

can use grammars to analyse a sentence and, from the Part of Speech (PoS) of its components, find a dependency grammar tree. Once we have extracted this information, we can try to produce a query in a formal language like SQL. Query languages are actually defined by formal grammars. This approach is attractive because, if we are able to translate the NLQ into a formal structure, the task is reduced to assimilate it to the formal grammar of the query language.

3.1.2 Empirical Approach

We have seen one perspective of language where we can understand it as a set of symbols that can be combined in order to get bigger structures. In other words, we can assume the functional character of language, that is, that the meaning of language expressions is a function of the meaning of its components. Instead, we can take a different approach based on pragmatic aspects.

"For a *large* class of cases of the employment of the word 'meaning' —though not for all— this word can be explained in this way: the meaning of a word is its use in the language" [37]. This is the way Wittgenstein proposed, in his Philosophical Investigations, the idea that the meaning of the language components is determined by its use in the overall linguistic domain. The philosopher proposed this idea as a reaction to his previous views of language as representation, that is concealed in a formal and closed system of meaning. In this new perspective, he sees language as another human activity with practical aims. And its meaning can not be analysed autonomously, isolated from the action that originated it.

In this view, we can not hope to find a perfect formalism that represents a language in its entirety. At most, we can discover, in Wittgenstein words, family resemblances between linguistic expressions. If we take this idea further, in more empirical terms, what we are looking after are patterns in language. Similarities in expressions give us similarities in their meaning. This is the inspiration of a second approach to Natural Language Processing, that employ a theory of meaning called distributional semantics. Such approach is based on what has been known as the distributional hypothesis: "linguistic items with similar distributions have similar meanings" [2].

This methodology is empirical in nature and implies a statistical orientation. The empirical aspect is evidenced by the use of large samples of language expressions, or corpora. These constitute the empirical world from which the statistical methods can be applied upon. In this category we find the different techniques of Information Extraction (IE) of statistical orientation, like named entity recognition, relationship extraction, terminology extraction, etc. Between the various statistical techniques we can find n-gram models, hidden Markov models and probabilistic context free grammars. Is beyond the scope of this work consider the details of these techniques.

In this category, we also consider the connectionist approach, that other authors consider separately. That is, the extraction of models based on neural networks. Even when these methods differ greatly from the classical statistical apparatus, the objective is the same. In both cases, we try to build a model based on previous experience, that we can use to make predictions.

[2] https://en.wikipedia.org/wiki/Distributional_semantics

3.1.3 Hybrid Approach

This third methodology combines the two previous trying to compensate the shortcomings of one or the other.

The symbolic approach is well-defined and predictable, but tends to be fragile. The complexity of human language, its inherent ambiguities and the numerous errors made, even by proficient speakers, makes it impossible in practice to capture all the possible productions in a set of rules. This is why, any group of rules that the designers can come up with, will fall short at dealing with the complexity of language. Furthermore, anything that is out of the scope of those rules is left behind. What is not handled by the symbolic apparatus is beyond the domain of the system, and this situation is hardly recoverable.

In the other end of the spectrum we find the statistical approaches. Their action domain is, let's say, continuous, with an associated distribution of probability, instead of discrete, like in the previous case. These techniques are more robust, in the sense that they can give a result (find a point in the continuum), even when is a bad one (low probability), for a wide range of situations (action domain). They depend totally on previous experience (data), and we have to assume that the phenomena coverage is sufficient. This is not easy in practice, so that data collection constitutes a problem on its own. The main disadvantage is that they fall short at modelling the complexity inherent in linguistic communication. Also, whatever they produce can not be mediated, that is, for example, we can not operate with the intermediate representations that a neural network has produced. Because of this, we can not adapt its behaviour but by adding more data. For the same reason, we can not explain why it has produced this result and not the other.

Hybrid approaches try to produce systems that take advantage of the robustness of the empirical approach, but with the flexibility of the symbolic approach. The exact way how to do this is not clear. There are many architecture designs possible that can combine these methods. If both of them live together in a single component, the designers must decide which strategies to follow in order to give priorities. This normally requires the development of heuristics.

3.2 Architectures

The literature is quite consistent in classifying NLIDB architectures [5, 25, 24, 27, 11, 2] in four groups according to the techniques they employ. The actual names and content of these groups change slightly depending on the author, for instance in [2], but in essence they are quite similar. In the following lines, I am going to expose a classification following the more traditional one proposed in [5], but keeping a look to the differences with other proposals.

3.2.1 Keyword Pattern-Matching Systems

One approach to understand the meaning of an NLQ is try to extract keywords present in it and match them with elements in a database. In this technique, first we perform a keyword-spotting step, where the terms are looked up in a dictionary built beforehand. This is why other authors define such systems as keyword-based systems, because "the

core of these systems is the lookup step, where the systems try to match the given keywords against an inverted index of the base and metadata" [2].

The set of keyword spotted is matched against a set of combination rules or patterns. For example, imagine an SQL table of countries with their capitals, we can think of rules of the form:

pattern: `... 'capital' ... <country>`
SQL: `SELECT capital FROM countries WHERE name='<country>'`

One of the most salient features of this architecture is its simplicity and, thereafter, its computational performance. They are also quite flexible. One can adapt existing patterns to new domains by changing the correspondences between keyword and database elements. But these systems have important limitations. They can not perform aggregation queries, for example, where a more fine-grained analysis of the NLQ is needed. Also, "the shallowness of the pattern-matching approach would often lead to bad failures" [5]. For instance, when one of the keywords corresponds with a term with a completely different meaning in another domain, the resulting query will be completely misguided.

In [2] the authors consider another category, that is an extension of this model. They call them *pattern-based systems*. Which are those that "extend the keyword-based systems with NLP technologies to handle more than keywords and also add natural language patterns." They also consider two types of patterns here: those that are domain-independent, like words indicating aggregation (eg 'by'), and domain-dependent ones, like concepts (eg 'good film').

3.2.2 Syntax-Based Systems

This type of systems use grammars to analyse the user's input. Here, "the user's question is parsed (i.e. analysed syntactically), and the resulting parse tree is directly mapped to an expression in some database query language" [5]. Using the syntax tree extracted from the input sentence, the system can map its elements to other elements in the database, like tables or relations. Having access to this kind of information about the user utterance, can also help in answer generation, making it easier to generate grammatically correct sentences that match the questions made by users, eg in time, person etc.

For this method to be successful, tables of the database normally have to be properly designed. It is difficult, sometimes impossible, to find general rules to map syntactic structures to any database schema. The problem here is that, in a relational database, the same content can be represented in many ways, all equivalent. It is therefore very difficult to port this kind of system to new domains or database structures. For this reason, "syntax-based NLIDBs usually interface to application-specific database systems, that provide database query languages carefully designed to facilitate the mapping from the parse tree to the database query." [5]

This type of system is called *parsing-based systems* in the classification in [2].

3.2.3 Semantic Grammar Systems

This type of system is similar to the former in the sense that it also employs grammars to analyse the questions. The difference is that it doesn't focus on the syntactic structure of

the phrase. In this case, it looks for semantic structures. Here, nodes have certain semantic content instead of just syntactic. One of such structures can be, for example, a conceptual map. Its leafs could be concepts reflecting objects in certain knowledge domain, and their parents categories of them. If we express such tree-structure in the form of a set of generation rules, then we have a semantic grammar.

Many times this technique is used to constrain the types of questions users can make. Limiting the language domain of accepted input sentences, can ease the question processing, and produce a sound answer. Another advantage is that grammar-based systems, as the authors of [2] call them, can have partial representations of the question. Thus, the system can give users natural language suggestions when they are typing their input.

Some authors consider this architecture difficult to port to other knowledge domains. The structure represented in a semantic grammar is normally very specific. A conceptual map for certain knowledge domain, animals for example, has different structure and different nodes that one made for a company organigram. This makes it very difficult to reuse grammar rules from one domain to another. In this sense, they can be regarded as less flexible than syntactic grammars.

3.2.4 Intermediate Representation Systems

Intermediate representation systems are those that use some meaning encoding extracted from the question. This codification normally comes in the form of a logical formula or a semantic tree structure. One of the main advantages of placing this intermediary language between the natural language query and the database query, is that we can abstract from the database structure. The intermediate representation constitute a coarse grained version of the query, at the level of its semantics. Often, the elements of this codification are concepts, or some high level representation of the world objects. This technique allows to gain an abstraction level from plain natural language expressions.

This type of system usually defines a series of modules specialized in certain operations that work on different levels of representation. The concatenation of such modules conforms a processing pipeline that produces the final answer. In figure 3.1 the reader can find a typical pipeline architecture with its modules. This is a quite old design, the literature shows schemas of this type before 1990. A more elaborate version of this architecture can be found in figure 3.2. This is an extended version showing the structure of an intermediate representation system and the modules that could make it up.

The operation of extracting the intermediate representation that we have mentioned before, only defines one of those modules. The semantic representation is independent of the natural language as well as of the database structure. This independence allows the system designer to connect with other modules that can help in understanding of the NLQ. One possibility could be adding a reasoning module that checks the consistency of the representation against some ontology.

The conceptual representation still cannot be used against any database in particular. Another module is needed to make this conversion. This is the responsibility of the query generator. This operation can be adapted to different database technologies thanks to the independence of the intermediate language and a translation module. It is possible, for example, to generate queries for a Prolog knowledge-base as well as SQL queries for relational databases. This is the design principle behind MASQUE [6], which communicates

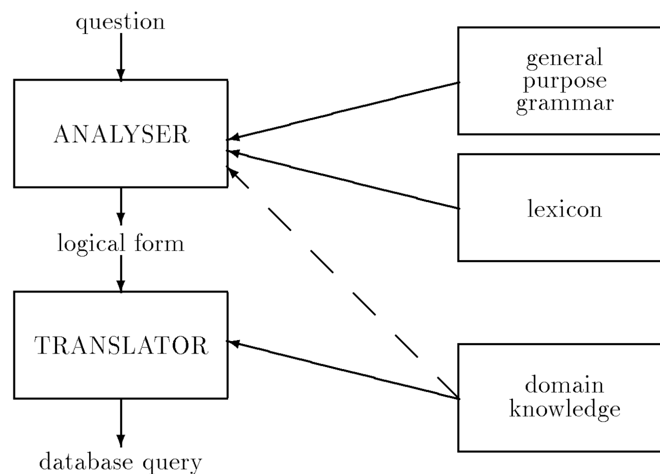

Figure 3.1: Typical pipeline architecture [14].

with a Prolog database, and MASQUE/SQL, that does the same for SQL queries. Both systems share the same infrastructure.

Once we have a query in some query language such as SQL, we can actually ask the database for the requested information. Finally, after querying the database, the system can manipulate the results obtained and answer the user. This last operation is the responsibility of the response generator. In this module we can also take advantage of the knowledge acquired in the analysis of the input. We can perhaps answer the users' questions in a particular way depending on the way they have formulated them. For example, we can make the phrase in the response to be linguistically consistent to the time, person, etc., in the question. This can make the response more natural to understand. Another possibility is, when the user is expecting a single item or a report, the system can adapt the generation to the expectations.

This architecture can be regarded as a de-facto standard. The other architectures are somehow integrated into this one. An intermediate language can be a syntax tree, like in syntax-based systems or a parse tree, in grammar-based systems. Even keyword pattern-matching systems can be interpreted as degenerated versions of this architecture. Some authors, like in [2], don't recognize this architecture as a type on its own, and rather assume it as a common pattern. Instead, the authors in this paper focus on the most salient methods employed inside the system.

3.2.5 Dialogue Systems

We have mentioned before that communicating with the user during the operation of the NLIDB can help in clarifying the question and improving the answer. The systems that can do this, incorporate architectural elements specific to dialogue systems. The classifications we have taken into account, don't consider this as a separate architecture. Whether it be because the architectures shown before can be included inside this one, or because dialogue systems follow their own structural principles, it is interesting, for what is to come, to present at least this design.

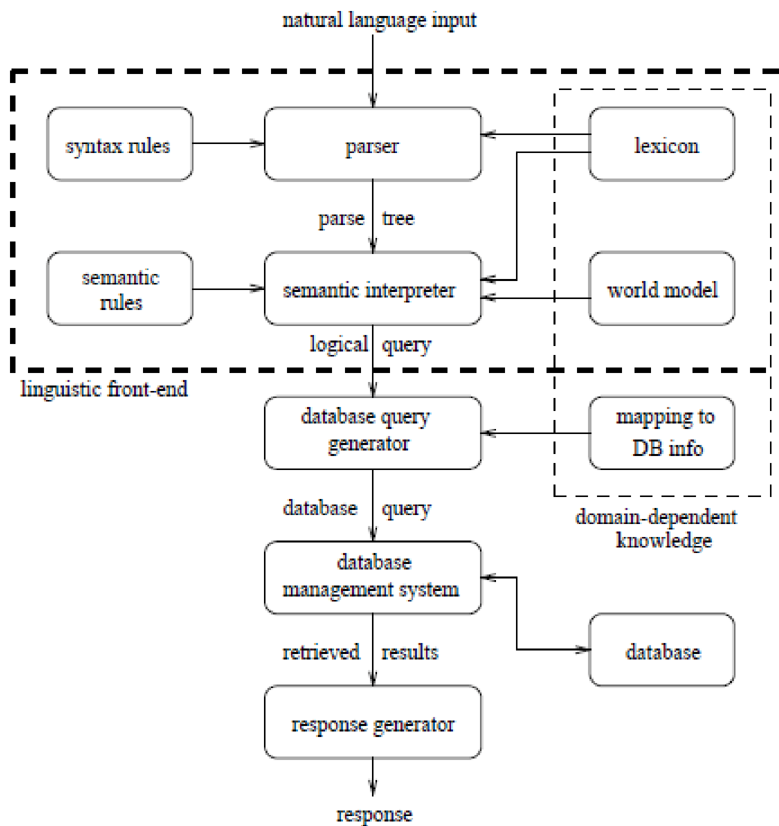

Figure 3.2: Architecture of an intermediate representation design [5].

The main difference with the other architectures we have seen so far is that the operation of the different components is orchestrated by a central component called the dialogue controller or dialogue manager. In the previous designs, the focus was on the modules dedicated to the interpretation of the input. Instead, in this case they are subordinated to the dialogue manager. In figure 3.3, the reader can find a schema of this type of architecture.

The dialogue manager is a type of reasoning module. Its purpose is to understand users' intentions and try to respond to them. It uses the understanding and generation modules as an interface with the user. These systems keep the state of discourse, that is, a record of the information the user and system have generated. This is used as context in order to improve the understanding and keep track of the state of the negotiation with the user. The database-specific modules, there may be, are used to find facts about the world and help to provide information to the user.

3.3 Challenges

There are several problems a NLIDB must face in order to perform properly, and they appear at all the levels of user input processing. Even when we consider the problem in its generality, the very formulation can bear witness to this difficulty. As is expressed in

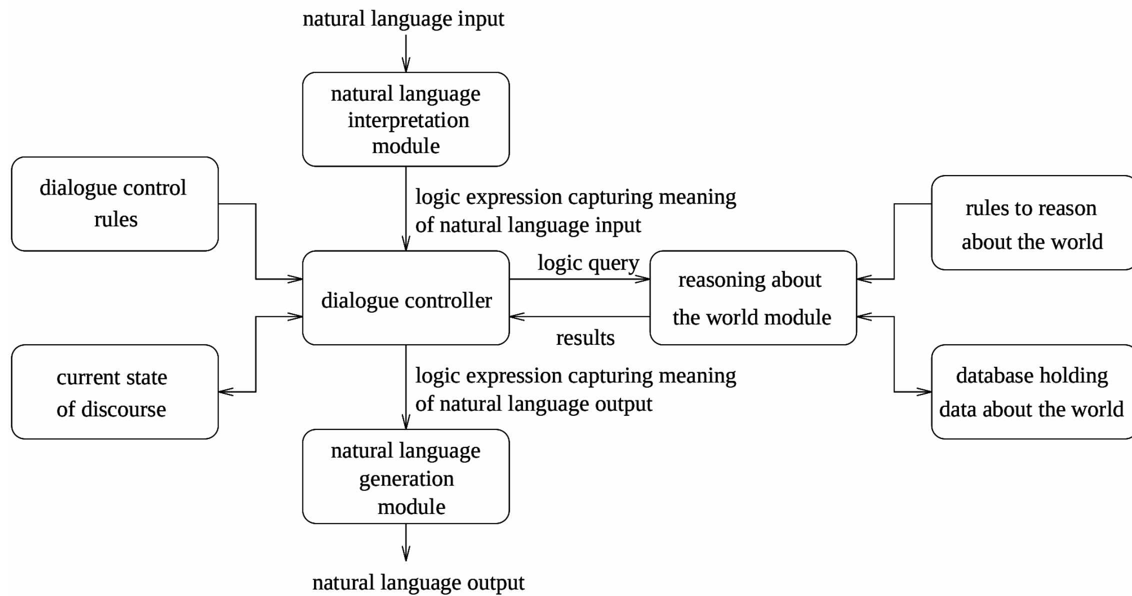

Figure 3.3: Possible architecture of a dialogue-oriented system [5].

[21]: "the goal for an NLIDB is to infer the user intent and this task is regarded by many as an 'AI complete problem'."

In this section we are going to introduce a series of challenges that this kind of system has been trying to solve since the early approaches. It will serve the purpose of building a bridge between the theory and praxis of NLIDBs implementation. At the same time, this will help us to define a set of particular objectives in the design. Several authors have made a recapitulation of historical challenges these systems had to face [5, 25], that we will follow in this exposition. In the next sections we present a list of the most relevant problems to be solved.

3.3.1 Linguistic Issues

Extra-grammatical Utterances

The first and most common problem in interpreting user utterances are misspellings and grammatical errors. Everyday use language contains plenty of these errors. Thus, a practical NLIDB cannot assume the sentences it will receive are free of typographic errors or syntactically correct, or even meaningful.

This problem is aggravated when the input comes from a spoken channel, thanks to the use of an Automated Speech Recognition system. The problems generated in the translation of speech to text are passed downstream the pipeline. The text produced by this type of system usually contains gibberish generated by noise in the audio. ASRs don't normally support separators, just alphabetic characters, which makes it difficult to separate phrases. Assuming that an ASR will handle well-formed sentences, can have fatal consequences for the overall performance of the system.

Scoping

Consider the sentence *"Find all the employees in the sales department over 30 years of age"*. It is immediately obvious to a person that *"in the sales department"* refers to employees and "over 30 years of age" refers to *"employees in the sales department"*. These two expressions are known as modifiers. The problem is that, syntactically, two interpretations are possible:

Find all the [[employees [in the sales department]] [over 30 years of age]]

Find all the [employees [in the [sales department [over 30 years of age]]]]

Determining which interpretation should be taken is called the modifier attachment problem. The resolution of this problem is not obvious to a machine. If the modifiers can be identified as elements by a parser, for instance, they will simply be at the same level, and two possible trees can be constructed based on them. In this situation there is an ambiguity that needs to be resolved somehow.

One option here could be the use of some sort of type checking. We can say that it doesn't make sense that a sales department have age. Or, does it? A sales department can be old. Even the type checking can be ambiguous. Another solution is the use of heuristics like the "most right association principle", that is, to prefer the rightmost argument of the modifier.

A similar situation arises with the use of quantifiers. Words expressing logical quantification like 'all', 'some' or 'any' also have a scope that is not always obvious. To make it more explicit, when there are several operators in one phrase, this problem consists in determining which one should be given a wider scope.

Consider the example sentence taken from [5]: *'has every student taken some course?'*. Here, two interpretations are possible. In the first one, each student can take the course they like. In the second, all students are supposed to take the same course.

As in the case of the modifier attachment problem, we can use some heuristics to solve this issue. One common heuristic is to preserve the left-to-right association of quantifiers. Or we can define an order of precedence between quantifiers, as it is done in the syntax definition of a language in a logical calculus that does not depend on brackets.

Ambiguous Terms

There are some terms that are inherently ambiguous in natural languages. One of such cases occurs in the use of conjunctions or disjunctions. This problem stems from the fact that the use of the word 'and' can sometimes denote conjunction, sometimes disjunction. Consider the phrase *'find every employee in sales and business risk departments'*. There are two possible readings. But we understand that the 'and' means disjunction in this case, and we want to know all the employees that work in sales department or in business risks department. We humans disambiguate this sentence thanks to the common knowledge that one person doesn't normally belong to two departments. Distinguish these two cases in a computer is problematic.

Another case of ambiguous terms are the so-called *nominal compounds*. Some compound names in English are semantically ambiguous. For example, 'city department' can both denote a department located in a city, or a department responsible for a city. As

in the previous case, we can use implicit knowledge to disambiguate, choosing the most common interpretation. However, implementing this in a computer again is challenging.

Anaphora

Anaphoric expressions are a well-known issue in Natural Language Processing systems. Linguistic expressions like pronouns (eg 'it'), possessive determiners (eg 'their') and some noun phrases like 'the owner' or 'these objects' are all examples of *anaphora*. They all implicitly denote some entity alluded previously in the discourse. They can be substituted by a fully expanded version of the sentence, where the referred entity is denoted explicitly.

If a machine interprets this type of sentence literally, it will certainly produce wrong results. For a machine to process a sentence like this, first it needs to detect the anaphoric structures, then needs to substitute it for the referred entity and, finally, use the expanded expression as normal input. There is a whole literature dedicated to this problem, with different strategies to mitigate this problem.

Ellipsis

Common language includes, many times, incomplete expressions. The meaning of which is derived from implicit knowledge about the context. Take the following two consecutive questions: 'how many goals were scored in the first half of the match?', 'and in the second?'. The last sentence is an elliptic one. We implicitly complete the missing information and understand a sentence like: 'and [how many goals were scored] in the second [half of the match]?'.

This type of substitution is a difficult task for a computer because it depends on contextual information that must be inferred. Many systems require the complete version of a query every time it is requested by the user. Again, this it is a common NLP problem and has its dedicated literature. We are not going to delve into the details of the possible solutions.

3.3.2 Interactivity Issues

Inappropriate Medium

This first interactivity issue asks whether language is an appropriate medium to interchange information with a computer. This entails a general challenge against the adequacy of language as a good communicator between people and computer systems. Some authors argue that human language is too verbose, or too ambiguous to suit the needs of an interface to information systems. Other types of interface, like graphical user interfaces, for example, are perhaps more suitable to translate user intentions to the well-defined form computers can deal with.

Multilingual

In order to widen the audience of an NLIDB system, it is interesting to produce interfaces with multilingual capacities. Many times systems are designed to support just English

as input natural language. Opening to other languages can prove itself challenging. The architecture must be such that the linguistic components can be translated into other languages. Otherwise, we would end up in a situation where one system has to be developed for each language. And, if we are in the need of developing domain-dependent components as well, the problem multiplies. It is thus interesting to study modular architectures that are flexible enough to support different languages without much additional development.

Empty Prompt Problem

The empty prompt problem [11], or sometimes called the "habitability problem", refers to the situation the user is confronted with when dealing with this type of system for the first time. This is similar to the situation novice users find themselves when they use a terminal prompt. Normally the system asks an open question like:

> *System*: What can I do for you?

This leaves users with little knowledge of the capacities of the system or the types of queries they can make.

Linguistic False Expectations

NLIDBs can suffer from false expectations in their linguistic capacities, which can hinder user experience. The first phenomenon that can produce this situation is the uncertainty in the linguistic coverage. This problem stems from the fact that the linguistic coverage of any NLIDB system is limited. Users find it difficult to understand the types of sentences the system can respond to [5].

Users tend to build mental models of what the system can understand or do, and abide to it. This means these models are not changed so easily. For example, if the language coverage is improved, this is likely to pass unnoticed. The user will assume the system does the same as before until something indicates the contrary.

Another phenomenon in this line is that, when a NLIDB can't answer certain question, it is unclear from the user's perspective whether this is because it doesn't understand the language or the underlying database doesn't contain the information needed.

This problem leads to situations where users try to rephrase the sentence with synonyms, hoping to find a term that is well understood. But the database perhaps can't find any information related to it because there isn't any. Or, conversely, they can assume the database doesn't contain any information regarding some term, but the problem simply is the term is not included in the lexicon. Again, this behaviour is translated into mental models that have their own inertia.

Capability False Expectations

We have already introduced a case of false expectations in the capacities of the system when we said that the database in the background can not have information at all about a particular topic. This is sometimes masked by the limited linguistic coverage problem.

Another issue in this line is that, when the system is able to answer some questions, can make some users assume it has reasoning abilities. Here we see mental models enter

the scene. Many NLIDBs don't include a reasoning module and are limited to the expressiveness of the back-end query language. Assuming intelligence normally leads to false expectations, and this can be frustrating to the user.

Response Generation

The generation of a response with the required information poses a new set of challenges. Assuming that the database query has been properly generated and the correct answer obtained, the way it is expressed could render it useless from the user point of view.

One of such cases is when the results returned contain codified information. For example, consider the user asks: 'what employees are older than 50 years old'. The query can produce a list of employee IDs. Even when correctly answered from the database point of view, the user may well be expecting a list of names.

Empty results or erroneous queries should be reported in an appropriate form to identify the source of the problem. There are several causes that may produce an erroneous response. Each case should be addressed differently. Also, the system may produce an invalid query, but it may be partially recoverable. In this case, dialogue strategies could help. The cause can lie in the linguistic coverage. In that case, if the system can detect this situation, it should be addressed differently. The empty set can be a correct result, and this should be prompted to the user.

Another situation arises when questions are interpreted literally, but the user has other intention in mind. A typical example of this are questions with binary form, that interpreted literally require a yes/no answer. But the user can formulate this type of question with the intention of receiving other information. For example, take the sentence: 'Is there a flight from Barcelona to Madrid tomorrow?'. The system may answer with a laconic 'Yes'. It could also save the user the obvious next question: 'Which ones?'.

3.3.3 Integration Issues

Portability

Following the re-usability principle, it is desirable for these systems to have some generic functionality that can be adapted in different circumstances. For example, the back-end database technology can change. We would like to be able to connect to different technologies such as SQL or SparkQL. This requires that the system incorporates translation modules. Inside a information representation model, we want to connect to different implementations. This is partly solvable through the use of query languages like SQL. But in the SQL family we find a group of dialects. Not all the expressions are supported in one dialogue or the other. These differences must be handled to make the system more generic.

Also, we would like to use the system for different database schemas. First, this may entail a change in the knowledge domain with a whole set of implications: in the terms used in the language specific to this domain, in the semantics in respect to the database elements. And second, even in the same knowledge domain, the same ontology can be expressed in an infinite number of schemas, all functionally equivalent. The system must provide with some configuration mechanism to adapt to the different database table designs.

We have mentioned before, when talking about the different architectures, the difficulties that these systems confront when it comes to portability. In the early times of NLIDB systems, these where tightly coupled with the underlying database technology and structure. Some architectures we have mentioned before, make it very difficult to port to different domains. We have seen some others, more modular in design, that are more easily adaptable. The need for portability has strong implications on the architectures that can be used.

Configuration

Adapting a system to a particular purpose is not a straight-forward step in NLIDB systems. A database technology, like SQL for example, comes with a schema definition and query language that can be used out-of-the-self. This allows users of this technology to build any schema they want and adapt it to any purpose. This is possible thanks to the formal nature of both the schema specification and query language. Whatever schema or query that is well-formed is guaranteed to produce results.

The situation with NLP-based systems is different. They are dependent of natural language, with all its ambiguities and inaccuracies. A previous step of configuration is necessary in order to adapt a system of this type to a particular purpose. The language needs to be channelled to that same purpose. If we assume that this step is unavoidable, then we want to make it as easy as possible.

One of the challenges regarding the configuration of an NLIDB is that it involves different domains of expertise. It is needed to know the particular structure of the database, which is normally in the field of the database manager. It is also needed to know the expressions used to refer to the elements in the database schema. This knowledge is linguistic in nature and requires different expertise. And, finally, an expert with the skills to express all this knowledge in the individual NLIDB technology, is also needed.

3.3.4 Lessons Learned

After about half a century of attempts of building a functional NLIDB, we have acquired quite some knowledge of what works and what doesn't. It is a good moment to recollect some of those findings before we present our proposal. I reproduce here a list proposed in [2] with some tips for a good design.

1. Use distinct mechanisms for handling simple versus complex questions: The way users formulate their questions depend on the complexity of the query. In particular, users tend to use keyword-like questions for simple queries. On the contrary, for more intricate queries, the questions tend to be more complex syntactically and grammatically correct. It is a good idea to have separate methods for handling each category.

2. The problem with subqueries: This is still a hairy topic, with difficult solution. The most common method to handle this problem is using parse trees.

3. Optimize the number of user interactions: Because the intrinsic ambiguity of natural languages, it is unrealistic to expect to get the final answer in one shot. Even

people aren't able to do this. It is best to interact with the user in order to clarify and resolve the ambiguities. In addition, the number of questions needed to reach consensus should be kept to a minimum.

4. Use grammar for guidance: The benefits stemming from the use of grammars are twofold. In the one hand, they contain useful information about the structure how questions are formulated, and this can improve the understanding. In the other hand, users get feedback about the questions that produce good results, and can adapt the diction to these patterns.

5. Use a hybrid approach: Both traditional NLP techniques and Machine Learning-based ones work best in combination. In particular, despite the promising results obtained with Neural Networks applied to certain tasks, it is still impractical to base the whole approach in this type of technologies. "Using a hybrid approach of traditional NLIs [Natural Language Interfaces] that are enhanced by neural machine translation might be a good approach for the future. Traditional approaches would guarantee better accuracy while neural machine translation approaches would increase the robustness to language variability" [2].

Chapter 4

Prototype Design Principles

4.1 Linguistic Approach

The architecture design of this proposal is aimed at a hybrid approach. This means that the architecture will allow the inclusion of modules based on both symbolic and statistical approaches.

The backbone of the system will be implemented using a symbolic approach. We intend that the processing pipeline be controlled and predictable. The spirit of this design is the contrary to that of an end-to-end neural network-based system. This approach will ensure accuracy for the most tractable cases. And will make the system deterministic, at this level, allowing at the same time its evaluation. This architecture will leave room for the inclusion of statistical techniques that can complement the functionality available, increasing its robustness. But it is beyond the scope of this work to measure the improvements that can be derived from the inclusion of such techniques.

The implementation will focus on delivering the necessary modules that conform together a functional NLIDB system. The implementation of the modules themselves will also be symbolic in nature.

If we attend now at the way the language refers to the external elements, whether they be database elements, or anything else in the outside world, we will be making assumptions over our conception of semantics. In particular, at the way we understand denotation. To make it explicit, we are going to assume an equivalence between reference and meaning. That is, nominal expressions in language, or representational tokens, will have a reference in the external world (the objects) in database elements. And this relation between expression and object defines the meaning of the former. The intention is making this relation explicit in the lexicon. For this, both components will be present in each lexicon entry. In the one hand, the expressions used to refer to database elements, and, in the other hand, some representation of the database element.

4.2 Proposed Architecture

This design will have a conversational spirit. We have exposed earlier the benefits that can be extracted from the interaction with the user. It will be useful to make use of structures and features of conversational systems. More than that, we can also use conversational

concepts to express internal representations. In terms of implementation, the prototype will be built on top a conversational system, although the focus will be on the NLIDB.

Some concepts related to the representations in a NLIDB system can be rephrased in conversational terms. Take for instance the idea of an intermediate representation language for a user query. We can now reconsider and reformulate it as a complex dialogue act in which the user is trying to retrieve some information.

The concept of *Dialogue Act* can be defined in general terms as "an utterance, in the context of a conversational dialogue, that serves a function in the dialogue"[1]. It is based on the idea of speech acts introduced by Wittgenstein [37]. They are typically classified in a taxonomy with the different functions they can perform in a dialogue. In our case, we are interested in two particular types, ie information retrieval and inform statement. These two acts contain respectively the pragmatic aspect of asking for some information and, in response, informing about the results.

We are now in position to reformulate the problem of a Natural Language Interface to Databases in conversational terms. An NLIDB system is thus an NLP system that takes a complex dialogue act containing an information query (of type information retrieval), and returns a dialogue act with the response (of type inform statement).

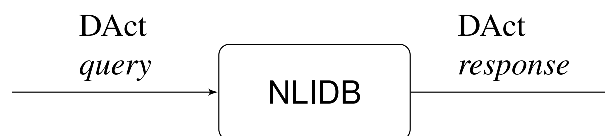

Figure 4.1: NLIDB dialogue act flowchart.

All along we have assumed that the pivotal point of the problem at hand is the construction of a system capable of translating natural language queries to database queries, and return the results. With the introduction of dialogue-specific concepts and infrastructure, we could well invert the terms and, instead, focus on the dialogue. If the subject were the dialogue, the NLIDB problem would be subsidiary to dialogue management. The functionality of interfacing with databases would be a component that provides with dialogue acts that can in turn be used by the dialogue manager. Changing the perspective this way, we find a useful component in dialogue systems, one that brings back information contained in a database, that can improve the interaction with the user.

We will structure the prototype according to the first interpretation. But it is interesting to note that, were the subject dialogue systems instead, the purpose and interpretation of NLIDB components would change, but mostly not their implementation.

If we now look at the single process of question answering, that is, how questions are processed, we will be attending to the architecture of the NLIDB part in the same vein as we have exposed it previously. Actually, this proposal incorporates aspects of almost all the architectures we explained in section 3.2.

If we look at the general structure, this proposal is an intermediate representation model. The objective of the question processing will be producing a semantic representation of an information query. In this prototype, we propose a simple information query representation, called semantic query (SemQuery). This is an approximation to a general representation of all the information may be contained in an information query. Here we

[1]https://en.wikipedia.org/wiki/dialog_act

will consider two elements in our ontology: targets and conditions. Targets are the denotations that have been identified in the natural language query. Conditions are the search restrictions imposed over those objects.

The third type of architecture we explained before was semantic grammar systems. This design also participates in our proposal to some extent. Based on the denotation model, that defines our semantic objects, we will build a semantic grammar from which the semantic query is constructed. The difference with the other semantic grammars is that the objects here are at the same level. It is a plain ontology where we don't consider explicit relations. We rely on the database schema to resolve such relations.

At the semantic level, we employ a chunk grammar. It will provide a robust method to handle simple queries. This type of grammar can be regarded as a pattern-matching system, like the first type we have exposed in the previous chapter. This grammar also looks for patterns, but it does so at the semantic level. The grounds of this type of grammar, where we read a sentence one chunk at a time, can be tracked back to the ideas presented in [1].

We could also consider a syntax-based grammar, like in the second architecture type. This would serve to handle more complex queries. But this type of grammar are out of the scope of this work. The interesting thing to consider is that other grammars could be inserted into this architecture, without changing the overall design.

4.3 Tentative Solutions

The main objective of the solution proposed is to define an architecture where the different problems that may appear when answering user queries, can be addressed incrementally without having to change the main design. We have presented a plethora of such challenges in section 3.3. With this prototype, we don't pretend to address all the issues at once. Although it will be interesting to show how to address some of them. At the same time, see in which places in the architecture are better handled.

Special mention deserves the problem of language as an inappropriate interface to information systems. We consider that the quality of the NLP systems to date haven't performed sufficiently well for users to consider them as a real alternative. This is more a problem of immaturity of the technology. It is perhaps too early to rule out this type of interfaces as some critics do. In addition, this problem is tied to the problem of information representation. A language interface only can access the information through some representation, that can not be the ideal when referring to it using language. In any case, we want to use existing technology, aiming at building useful systems. For this reason we employ relational databases as the back-end technology for data storage.

In table 4.1 the reader can find a list of tentative solutions to the problems presented in section 3.3. We try to localize the problem in one of the components in our architecture and explain how could be addressed. Broadly speaking, the solutions proposed lie in the following ideas. We want to confine the representation problems in the lexicon and, at the same time, ease the configuration having to look into a single component. Also, we would like to make use of the dialogue framework features to cope with another set of problems. When it is useful to communicate with the user to disambiguate or clarify, and to keep context and memory of what has been said. In general, the architecture should allow the inclusion of more sophisticated components that can address specific problems.

	Problem	**Module**	**Solution**
Linguistic Issues	Extra-grammatical utterances	Spelling corrector, Chunk grammar	Correct the spelling based on the lexicon; use a chunk grammar to make the system resilient to noise.
	Scoping	Dependency grammar	Use grammar to find dependencies and apply heuristics.
	Conjunction / disjunction	Error handling	Ask the database and use heuristics to select a meaningful query.
	Nominal compounds	Lexicon	Fix the meaning in the lexicon.
	Anaphora	Grammar, Dialogue history	Detect the anaphoric expressions and substitute them using the dialogue history.
	Ellipsis	Dialogue manager, Dialogue history	Detect missing information, try to fill it with the dialogue history or ask the user.
Interactivity	Multilingual	Lexicon	Only the database lexicon should be translated.
	Empty prompt	Dialogue manager	Answer questions regarding capabilities.
	Linguistic false expectations	Dialogue manager	Relay on the interaction with the user to clarify linguistic coverage.
	Functional false expectations	Dialogue manager	Similar to the above, clarifying the knowledge domain.
	Response generation	Multi-modality	Use an appropriate medium to the type of answer.
	Codified information	Lexicon	Denote the name of the object in the lexicon instead of the identifier.
	Literal interpretation	Grammar, Dialogue manager	Detect binary questions, extend the response to include more information.
Integration	Portability	Skills	Modular architecture with an NLIDB framework plus per-database skills.
	Configuration	Lexicon, Grammar	Focus customization on the lexicon and some database-specific grammar rules.

Table 4.1: Summary of tentative solutions to NLIDB problems.

Chapter 5

Implementation

In the previous sections, we have introduced the ideas on which we are going to base the development of our prototype. Here, we are going to delve into the implementation details. In the following pages, we will present the different components, their function and the technologies used to build them.

The specific use cases are described in the next chapter. And we will refer to those examples at some points in this description. Yet it might be useful to the reader to get a grasp of the processing pipeline first, taking a quick look to section 6.1.

5.1 General System Architecture

The system is divided in three major functional modules: the interpreter, which extracts the meaning of the user question, the query generator, which translates this to the database query language and retrieves the information, and the response generator, which produces the final answer. Each of these subsystems is compound of other modules dedicated to more specific tasks.

Overall, the system forms a processing pipeline, where the data is transformed step by step as it passes through the chain of modules. In figure 5.1, the reader can find a schema of this pipeline, with all the components involved. Also, the data processing is annotated in the flow arrows, indicating the different transformations the data experiment along the way.

The technologies employed for this prototype, in general terms, are Lekta [29] for the linguistic part, Haskell for data processing, and SQL as database management system. Other technologies and frameworks will be mentioned when they appear during this description. Lekta technology provides an NLP pipeline, with its own programming language to describe grammars and define rule-based systems. It also ships with a dialogue framework called Fluency. Both form the dialogue framework we mentioned before, on which we are going to construct our prototype.

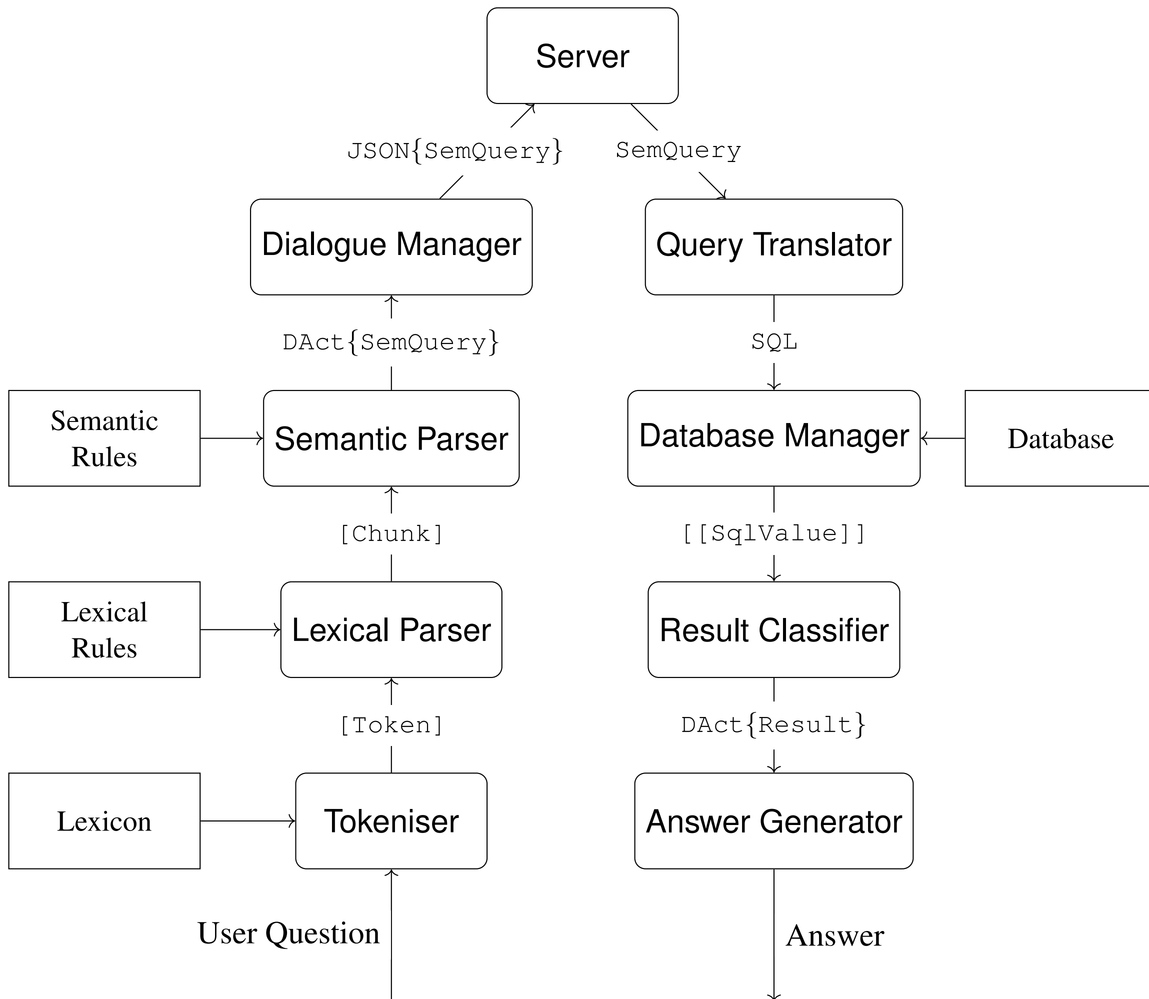

Figure 5.1: Processing pipeline with all the components and data flow.
Note: The annotations in the arrows indicate the pseudo-types of the data after a process operation. Square brackets denote lists, and curly brackets indicate the contents of another type.

5.2 Interpreter

The interpreter is responsible for processing the user's natural language query and transforming it into a dialogue act containing its semantic representation. Schematically:

$$Query \rightarrow DAct\{SemQuery\}$$

5.2.1 Lexicon

The terms in the lexicon contain the vocabulary that defines the linguistic coverage of the system. In the case of database-related terms, it also defines the mapping with the database elements. We do this following the denotation model we have explained before. Each lexical entry is defined by a type, and contains the linguistic tokens as well as the contents for that particular type.

Domain-independent Lexicon

The query lexicon is the generic vocabulary contained in the subdomain of language dedicated to information requests. It is domain independent, and can be reused across applications. This lexicon contains terms such as functions, comparison operators and logical connectives. As any other lexicon, this component depends on the language.

The first type of term that we find here is the information function (`lexDBFunction` type). This type of term express operations that can be applied over some piece of information. These normally map directly to SQL functions. Following, the reader can find a list of such functions with term samples:

Function	**SQL**	**Lexicon**
Counting	`COUNT`	'count all', 'total number of', 'how many'
Addition	`SUM`	'sum', 'compute the aggregate of'
Average	`AVG`	'average', 'find the arithmetic mean of'
Minimum	`MIN`	'minimum', 'the lowest'
Maximum	`MAX`	'maximum', 'the largest'

The second type of term we are going to consider are comparison operators (`lexDBOperator` type). In this class we find terms used to specify comparison of elements. The implementation is similar to that of functions.

Operator	**SQL**	**Lexicon**
Equality	=	'equals', 'is equal to'
Inequality	<>	'doesn't equal', 'is not the same as'
Strictly greater	>	'greater than', 'above'
Greater	>=	'greater than or equal to', 'at least larger than'
Strictly lower	<	'smaller than', 'below'
Lower	<=	'less than or equal to', 'at most below'

Domain-dependent Lexicon

There are three lexicographic types defined that can be used to define the correspondence between natural language utterances and database elements. These are: topics, targets and values. It might be clarifying to see how this correspondence is established in a real case, having a look at section 6.1.

The first ones are *topics*, with type `lexDBTopic`. These are simply utterances that refer to some topic or knowledge domain. They can be useful to define the current subject

under discussion, in a multi-domain application, and can define the context. A topic can refer to a specific database, thus can help steering database connections.

lexDBTopic A knowledge-domain or topic
```
("in the Euro cup 2016", lexDBTopic, 'Euro2016')
```

Targets, with type `lexDBTarget`, define a correspondence between utterance chunks and structural elements in the database. The structural part is defined as a string containing the path in the database schema. This format follows the SQL convention of designing elements with a dot-separated list of structural levels. With this type, we can first refer to a table, when the object is represented by the rows of that table in general, and they don't have a column with name (eg 'substitution'). Or a column in a particular table, when we refer to something more specific that has a dedicated column (eg 'goal minute'). Or, finally, a column without parent table, similar to the previous, but for objects that can be in different tables or qualifies objects using a column (eg 'half', can be 'first half goal', 'first half substitution', etc).

lexDBTarget A mapping between an utterance chunk and a database element.

- Table: `("substitution", lexDBTarget, 'player_in_out')`
- Column in a table:
  `("goal minute", lexDBTarget, 'goal_details.goal_time')`
- Column without table: `("half", lexDBTarget, '.play_half')`

The third lexicographic elements we can declare are *Values*, with type `lexDBValue`. These also refer to a target but, in addition, they define a specific value for that target. This type can express conditions. Or convey elements that, for implementation reasons, are represented in the database as classifications by certain value. For example, we can designate match victories in a table of matches, using a column to classify them by certain value (eg `'V'`). Note that this is an implementation decision; all 'victories' could be set apart in a different table, and we could refer to this table instead. A third use arises when we extend this idea to unique elements which are *classified* by a unique value. The value is represented by a serialized SQL version of the value. For instance:

lexDBValue A database target with a defined value.
```
("victory", lexDBValue,
 (lexDBTarget:'match_mast.results', SQLValue:'\'WIN\''))
```

5.2.2 Lexical Parser

Based on the lexicon we have defined, that specify meaningful chunks, we can now build a grammar that elevates these chunks to a semantic level. In this type of grammar the rules are pretty simple. Because we are considering utterance chunks that are more or less meaningful by their own, these rules basically perform a mapping from lexicon to semantics and pass through the information contained.

Semantics in this implementation is populated by a limited set of entities. The types of such entities are the following:

DBFunction A function on some piece of information. It contains the name of function to perform. This is the semantic level of the utterances that express operations, which we have explained in the lexicon.

DBOperator An operator between pieces of information. It contains the name of the operator it refers to. This is the semantic content of lexical operators.

DBTarget Semantic representation of an element in a database, containing the path in the database structure to that element.

DBValue The semantic content of a value in the database. It contains both a `DBTarget` and a `SQLValue`. The second is a serialized representation of the value pointed by the target.

DBCondition This is the semantic representation of a condition in the retrieval of some information. It represents some restriction on the domain of objects to fetch, the limiting criteria. That is, a generalization of what we understand as a condition predicate in the where-clause of a database query. Conditions are compound by three elements:

- A target as defined before. Denotes the subject of the condition.
- An operator as defined before. Expresses the type of comparison to be made.
- A value, in its SQL serialized version. Contains the value against the subject is compared to.

As we mentioned before, the grammar rules defined here are straightforward. A typical rule of this kind looks like this:

```
(NLIDB_lex_DBFunction :
    [ DBFunction -> lexDBFunction ]
    {
        ^.FunctionName <- #1;
    }
)
```

This is Lekta syntax to express parsing rules. The expression between square brackets is the production rule in a context-free grammar. The arrow differentiates the left-hand side (LHS) of the rule and the right-hand side (RHS). Its elements are the types we have defined before. In this particular case, `DBFunction` is the LHS of the production rule, `lexDBFunction` is the RHS. Accordingly, this can be read as: every utterance chunk that expresses an information function is an information function at the semantic level. The one-to-one relationship between lexicon and semantics is characteristic of chunk semantics, where its elements are meaningful by themselves.

The body of the rule, the part between curly brackets, establishes the operations that define the contents of the LHS element. The caret expresses what is going to be taken up in the creation of the new instance of the LHS type. In this particular case, we define the name of the function by taking with us the classification that we had defined for operation expressions. The reader can find a practical example of this in section 6.1.

The majority of the rules in this grammar are the same, *mutatis mutandis*. But we can also consider other surrounding words and expressions that give us higher certainty that we are pointing to the right semantics. At the same time, we reduce parsing ambiguities by increasing the token extension of the rule. These are preferred by the Lekta parser, thanks to a built-in heuristic. This way, we claim for that semantic content a bigger part of the sentence that otherwise another rule could take.

Let's consider an example of utterance expressing a condition. There are some words normally used to introduce such conditions (like 'in', 'within', 'in the'). We can write a rule that considers these words, and swipe them from further parsing at the same time. If we find some of these words followed by a `DBValue`, we probably are looking at a condition. The rule would look like this:

```
(NLIDB_lex_Prep_DBValue :
    [ DBCondition ->
        < lexPrepWithin | lexPrepIn > lexDetThe? lexDBValue ]
    {
        ^.DBTarget.TargetPath <- #3.lexDBTarget;
        ^.DBOperator.OperatorName <- 'EQ';
        ^.SQLValue <- #3.SQLValue;
    }
)
```

The lower-than and less-than symbols define a block of alternatives separated by the vertical bar. Also, the question mark tags an element as optional. This is just syntax sugar to compress several rules into one. We can read the head of the rule as follows: the preposition 'in' or the preposition 'within', followed by (optionally) the determinant 'the', followed by the expression of a database value, is a semantic condition.

Let's now look at the body of the rule. When we use a defined value in the database as a condition, we implicitly say that the comparison to be made is the equality. Because we are referring to that value and not other. That is the reason why we explicitly set this condition.

The semantic elements constructed by this grammar are then promoted as parameters for the next level grammar.

5.2.3 Semantic Parser

Once we have performed the first level of parsing, we have at our disposal a set of parameters of different types. We can now use them to extract the pragmatics contained in the user question. That is, to find out the intention of the user as well as the contents of those intentions, such as what type of information is he looking for, having which conditions, etc.

For this reason, we employ a second level grammar that establishes the pragmatic elements contained in the question from the semantic elements of the query. The constituents we are looking for, at the pragmatic level, are called dialogue acts (DAct) in dialogue systems [20]. We could employ a different nomenclature, but it is useful for our purpose to reuse this one, specially considering that our infrastructure uses a dialogue framework.

We will thus define a grammar that takes semantic elements and produces dialogue acts. In the terminology of Fluency [29], this is called *Pragmatic Mapper* rules.

The first part of the user intention we are interested in is the purpose of querying information itself. This is perhaps too obvious in the case of monolithic NLIDB systems, where the only type of questions that can be answered are natural language queries. But, if we want to generalize the problem, and consider this functionality together with other *skills*, we have to detect this first. In the taxonomy defined in Fluency, this type of dialogue act can be classified as a request to perform an action, which in turn is querying information. We can codify it this way:

DAct:Request/Action/NLIDB Dialogue Act that indicates the action request of making a natural language query.

There are some expressions that indicate we are before a factual question of this kind. Expressions like ‘find’, ‘prepare a list of’ or ‘make a report with’. All prepare the interlocutor for an information request. At the same time, they provide hints on the expectations about the format of the final response: a single value, a list, a table, etc. We can use these expressions to detect the intention of making a request of information retrieval, therefore raising the corresponding DAct.

But the intention of querying some information must be followed by some content. We define two types of contents that can be provided for the fulfilment of a query: targets and conditions. In our taxonomy, they are actions performed by the user to inform of something, and that something is the provision of a query parameter. These parameters constitute the units in which an information query can be divided. The classification of these dialogue acts is as follows:

DAct:Inform/Parameter-provide/target Dialogue Act containing the provision of a query target.

DAct:Inform/Parameter-provide/condition Dialogue Act containing the provision of a query condition.

These dialogue acts are constructed based on the parameters provided by the previous grammar, but transformed into pragmatic elements, that is, dialogue acts. To accomplish this, we use a second level grammar. The right-hand side of these rules are the semantic components obtained in the previous parsing phase. That is, the self-meaningful chunks we have extracted. The left-hand side of the rules are dialogue acts with its contents. The general rule schema looks like this:

$$DAct_1 DAct_2 \ldots \rightarrow Chunk_1 Chunk_2 Chunk_3 \ldots$$

Grammars of this type are context-sensitive in Chomky’s classification. They are quite complex to parse in general, but we are going to limit it to terminal symbols in the RHS and non-terminal in the LHS. To resolve ambiguities, the following heuristic is employed: prefer longer RHS conditions, then prefer the first defined. The implementation of this parser is provided by the Fluency dialogue framework that ships with Lekta technology.

The most important parameter that a user can provide is what we call a *target*. We don’t consider a query complete, unless at least one target is given. As we said before,

targets denote elements in the outside world, elements of the database in our particular case. At this point they are fully qualified. If functions are present, then they are saturated (in Fregean terms), that is, they are applied to something. This means that, if the target contains some functions, this will be included as part of it. For this reason, at these point we find rules that transform semantic functions followed by semantic targets into a single pragmatic target: $DAct\{f, x\} \rightarrow Chunk\{f\}Chunk\{x\}$. A concrete example of such a rule can be found in section 6.1.

The second components we consider as pragmatic parameters are conditions. These restrict the search domain of objects. As in the case of targets, they must be saturated, in the sense that, if there is a filter, the filter refers to something. This means that all the fields in the filter must be filled. That is, they will always include a target, an operator and value to compare with. The reader can refer to section 6.3 to see this type of rule in action.

5.3 Database Query Generator

In general terms, this subsystem takes the semantic representation, transforms it into the underlying relational database query language (SQL) and, after querying the database, handles the resulting table back.

$$DAct\{SemQuery\} \rightarrow [[SqlValue]]$$

5.3.1 Back-end Communication Channel

Format

Lekta technology uses a callback mechanism to communicate with back-end systems. This is a generic method that allows the programmer to plug in any technology he may need. To do that, one must implement the appropriate plug-in. A default implementation is provided for connecting with web services using GraphQL [15].

The communication between the Lekta application and the back-end system is mediated by a GraphQL schema that defines the `nlidb` operation. It receives, as input, a representation of the semantic query generated previously. In turn, it responds with the results of the database query. The operation is defined like this:

```
type Query {
    nlidb(semQuery: SemQuery!): String!
}
```

Note that the database response is defined as a `String`. It is given as a serialised version of the SQL table the DBMS generates in response. The only argument of this operation is the semantic representation of the query, defined as a `SemQuery` type. It is very similar to the output of the second level grammar. Indeed, the application takes a previous step before sending the request. Before that, it has to be transformed into the JSON format defined by the schema. Bellow, it follows the full definition of this type:

```
"Semantic representation of an information query"
input SemQuery {
    command: String!
    targets: [Target!]!
    conditions: [Condition!]!
}

input Target {
    argument: String!
    function: String
}

input Condition {
    subject: String!
    operator: String!
    value: String!
}
```

This is a simplistic version of an information query. It is a first approach to a general (we could say vast) problem. In this representation, we have, first, a required `command` that defines the type of operation we want to perform. In this prototype we will limit ourselves to information retrieval (no updates or others). Second, we have a list of `Targets`. These are formed by a required `argument` and an optional `function` that is applied over it. `arguments` are the paths to objects in the database schema. And, finally, we have an optional list of conditions. We declare it as mandatory, but we can send an empty list when necessary. Similar to what we have said with the definition of semantic types before, we have in the `Condition` a `subject`, an `operator` and a `value`. Their meaning is the same as what we exposed before, just in different format.

Server

The server itself is implemented in Haskell using *Scotty*[1] for the web server and *Morpheus*[2] for the GraphQL infrastructure. What we have to implement is the corresponding resolver for the `nlidb` operation. The function definition is the following:

```
resolveNlidb :: NlidbArgs -> IORes Text
resolveNlidb = resolver . nlidbBackend . semQuery
```

Interpreting the signature of the resolver, we discover how the GraphQL request is processed. The resolver takes the arguments of the request and returns a string wrapped in some monadic effects. The expanded version of the `IORes` monad is `(ExceptT String) IO`. The type `ExceptT String` is a monad transformer that adds exceptions of type `String` to another monad, `IO` in this case, the monad of input / output effects. Now, reading backwards the body of the function, we first have `semQuery` which just extracts the field with the same name from the arguments. This

[1] https://github.com/scotty-web/scotty
[2] https://morpheusgraphql.com/

is passed onto our back-end function. And then we have `resolver`, which is a functor that takes our back-end function and lifts it to the mentioned monad, adding the GraphQL schema validation, the response generation and the rest of the GraphQL infrastructure as transformation of the `IO` monad.

Therefore, all we have to do is to define the function that does our back-end stuff, leaving the rest to the `resolver` function. Here is the definition of the `nlidbBackend` function:

```
nlidbBackend :: SemQuery -> IO (Either String Text)
nlidbBackend semQuery' = do
  let sql' = toSql semQuery'
  sqlBackend sql'
```

This is the function that takes the semantic representation of the query, transforms it into an actual SQL query, and evaluates it in the database. If everything went fine, it returns the results produced, otherwise an error string. The important functions to look into are `toSql`, that transforms the semantic query into SQL, and the `sqlBackend`, that evaluates it.

5.3.2 Database Query Translator

This is the module responsible for translating the intermediate representation to an actual database query. The complete syntax for a select statement can be quite intricate[3]. It also has variants or dialects that differ from one back-end technology to another. For this prototype we are considering standard SQL with a limited set of features.

The query translator is implemented as a Haskell class called `ToSql`. This class represents all the types that can be converted to SQL. It defines a single method called `toSql` which translates certain type to its SQL representation. The definition of this class and its most basic instance reads like this:

```
module Data.Conversion.ToSQL where

type SQL = Text

class ToSQL a where
  toSql :: a -> SQL

instance ToSQL SQL where
  toSql = id
```

The heaviest work in this module is done when transforming the semantic query, with type `SemQuery`, to its SQL representation. This process is implemented instantiating the class `ToSQL` for this particular type. The main body of such function looks like this:

[3]See, for instance, the MySQL select statement definition: https://dev.mysql.com/doc/refman/8.0/en/select.html.

```
instance ToSQL SemQuery where
  toSql (SemQuery command targets conditions) =
    command <> "␣" <> toSql targets
      <> resolveRange targets conditions <> toSql conditions
```

This function constructs the final string containing the SQL query. The structure of an SQL query becomes apparent in the body of this function. It is formed by a command followed by some columns extracted from the targets. Then goes a from-clause that defines the tables where the data is located, plus the join paths. And finally a where-clause with a set of conditions that defines the rows to retrieve.

Some elements can be translated directly into parts of the SQL query. This is the case of the field names that follow the statement operation. We implement this instantiating the class `ToSQL` for the list of targets. Each target is compound of an optional function and an argument. We have to take this information and transform it into its SQL representation of the form `FUNCTION(ARGUMENT)` or just `ARGUMENT`. The implementation for the `Target` type is the following:

```
instance ToSQL Target where
  toSql Target
    { function = func
    , argument = arg
    }
    = case func of
      Just f  -> f <> "(" <> column <> ")"
      Nothing -> column
    where
      column = case columnOf . pathOf $ arg of
        "" -> "*"
        x  -> x
```

Not all elements used in our intermediate language can be translated directly to SQL. Because we are aiming at a more general semantic representation of information requests, we can not map everything directly. One notable exception is the resolution of the from-clause and the inference of joins. This is done in a dedicated function called `resolveRange`. This function is responsible for finding the from-table as well as resolving joins for a particular table schema. In the following snippet of code, the reader can find the Haskell definition of the types involved, as well as the declaration of the functions involved in this task.

```
-- SQL path, ie 'schema.table.column'
type Schema = Text
type Table = Text
type Column = Text
type Path = (Schema, Table, Column)

type Relation = (Path, Path)
type RelationGraph = [Relation]
type RelationPath = [Relation]
```

```
-- Resolves from-clause and joins, and translates them into SQL
resolveRange :: [Target] -> [Condition] -> SQL

-- Generates the SQL join expressions from a relation graph and
-- a list of tables
resolveJoins :: RelationGraph -> [Table] -> SQL


-- Resolves the relation, in a relations graph, between two
-- tables, if possible
findRelation :: RelationGraph -> Table -> Table
                -> Maybe RelationPath
```

We assume that the schema defines primary keys and foreign keys for all relations. When this is the case, we can query the database to find all the relations. In appendix A.2, the reader can find a query to get all relations in a PostgreSQL database. The relations are defined as a list of pairs of keys. These define a relation graph we can use to search for paths. That is, find what tables are related to another through some combination of joins.

The implementation shipped in the prototype is rather naive. A better approach to the inference of query join paths can be found in [8]. The important point to note here is that another implementation with the same signature will work just fine. Thanks to the definition of type functions in Haskell, and the modular architecture of the code, one can just implement a new function with the same signature and the program flow won't change.

5.3.3 Database Communication Layer

The communication with the database management system (DBMS) is done using ODBC (Open Database Connectivity)[4]. For its implementation, we use the Haskell database connectivity framework (HDBC)[5], which defines a common API for the connection between relational databases and Haskell applications. For ODBC, we use HDBC-ODBC[6], which provides a database back-end driver for this connectivity protocol.

The system expects some drivers to be installed in the host system and some configuration for the connection to be successful. This project is thought primarily to be used with MySQL/MariaDB and PostgreSQL databases. The host system must have drivers for these two types of databases, plus a `odbcinst.ini` configuration file. In **??** the reader can find an example of this configuration file.

ODBC uses connection strings to define a connection to a specific database. It contains information such as the driver to use, the domain name and port where the database can be accessed, the name of the schema, user and password, and perhaps other information. This string is built automatically using the environment in a configuration cascade model. First, the systems looks for environment variables, if they are not defined, it looks for those variables in a `.env` file. Finally, if neither of those were found, it uses a hard-coded default value. This makes it easy to switch between environments and have the

[4]https://en.wikipedia.org/wiki/Open_Database_Connectivity
[5]https://github.com/hdbc/hdbc
[6]https://github.com/hdbc/hdbc-odbc

connection automatically configured from the start. An example of a connection string for a PostgreSQL database looks like this:

```
Driver=postgres;Server=127.0.0.1;Port=4000;UID=postgres;PWD=***;
```

The actual query is performed by a function called `query`. This function takes a SQL query as a `String` and produces a table of SQL values with some side effects, ie has type `IO [[SqlValue]]`. The full definition of this function is:

```
query :: String -> IO [[SqlValue]]
query sql = do
  connStr <- getOdbcConnStr
  conn <- connectODBC connStr
  quickQuery conn sql []
```

5.4 Response Generator

This is the smallest subsystem of the pipeline. It receives as input the results table produced by the database and, ultimately, it will produce the answer that the user sees.

$$[[SqlValue]] \rightarrow Answer$$

5.4.1 Response Classifier

The response generator works in the context of a dialogue manager (DM). Once the database produces a result, it is returned to the DM. Then, the DM takes the results and classifies them. Depending on the pattern of the table returned, the response will be of one type or another. This is important to present the answer in an appropriate format. There are four classes taken into account in this system:

db_result_value A single value. This is the typical response for factual questions, where a single piece of information is expected.

db_result_table A table. Normally answered when some sort of list or report is expected.

db_result_empty An empty table. The query was successful, but it produced an empty result. This usually means the conditions where too restrictive and no element satisfied them.

db_empty_response No response. This is produced when the database returns an empty result, whether it be caused by an incorrect database query or a technical error.

5.4.2 Answer Generator

Once classified, the database emits a dialogue act to inform the user. In particular, the type of this DAct is `Inform/Statement/<result_type>`, where `<result_type>` is one of the list above. The contents of this dialogue act will be a cleared up version

of the data returned by the database. By cleaning up, we mean removing unnecessary wrappers in this data.

The actual response generator is the component that takes this output dialogue act and produces the answer. This separation of concerns gives us more flexibility to adequate the format of the answer to the format of the data. That is, we gain multimodality. At the same time, this DAct can live together with others generated by the dialogue manager, other DActs that can produce other information interesting to the user.

The implementation in this prototype is more focused on the multimodal aspect, rather than in producing linguistically correct answers. For this reason, for a single value, we present the user with a neutral phrase preceding the answer, independent of the scalar type of the piece of information, and of correlations with the question. Also, for the two degenerated cases, `db_result_empty` and `db_empty_response`, we produce a predefined message. An example of a single value answer could be:

> *User*: Find the number of goals scored in the whole tournament.
> *System*: The answer is: 108.

In the case of having to present a table, we prepare the data so it can be processed by other interfaces. To accomplish this, the system takes advantage of the *Output Context Data* (OCD) feature of Lekta [29]. This is an output communication channel devised to plug the response to another system. What the system does is including the data, formatted as widget JSON object, in the `commands` field of the OCD. This way, another system, a mobile application for example, can process the data and generate the appropriate widget in the screen.

> *User*: List goal author and goal minute in the first match.
> *System*: See the results on your screen.

```
{
    "commands": [{
        "type": "widget",
        "subtype": "table",
        "data": [
            ["89","Dimitri Payet"],
            ["57","Olivier Giroud"],
            ["65","Bogdan Stancu"]
        ]
    }]
}
```

5.5 Project Organisation and Deployment

We have seen the definition of the different modules that are part of the system. But this doesn't reflect precisely the way the project code is organised. It does not describe the deployment modules either. Therefore, it is necessary to describe the project from the point of view of the technology consumers (developers), answering the question of what they need to do to adapt the technology to their purposes. And, in the other hand, from the point of view of the development operations engineer (DevOps), describing what is needed to take the project into production.

5.5.1 Skills

Domain-Independent Framework

Regarding code organisation, we want to group the domain-independent functionality into a shared infrastructure. This defines a framework that will be shared between all NLIDBs. In particular, it is implemented as set of skills. Skills are the Lekta-idiomatic units of functionality in which projects are organised [29]. These units are independent functional modules that mirror the multiple capacities of a competent speaker, in the sense of the chomskyan notion of linguistic competence.

The common framework is thus composed of several skills that include the general lexicon that refers to query functions, comparison operators, etc. That is, the domain-independent linguistic capacities of the system. Ideally, these capacities would be multi-lingual, with translations in many languages.

This framework also contains the domain-independent semantic rules that can be applied to any lexicon defined. This part of the system is already language independent.

Database-Specific Skill

What the technology consumer would need to implement is just a database-specific skill. In that skill, mainly in the lexicon, the consumer would define the linguistic domain for the database to use. At the same time, the correlations between language and database elements are codified in the lexicon entries. With the use of the types defined in section 5.2.1, the programmer can define the database structure and how the information is stored in it.

In addition, when necessary, the consumer can define an extra set of semantic rules. The rule execution model allows the inclusion of further rules that can complement the processing made by the common framework. Such rules can be useful when the semantic chunks given by the framework must be treated in a particular way. This could be the case of dates, numbers, etc., that the framework's grammar already handles, and need to be given a special treatment to match the database structure.

5.5.2 Micro-services

Containers

The project is prepared for deployment using a micro-services architecture. For this, the different components are organized in containers using Docker[7]. A normal NLIDB project will define three containers:

- Lekta web application: NLIDB framework plus the user-defined database skills.
- Back-end server: GraphQL server and database connectivity.
- Database: a container with the actual database, with its corresponding engine.

[7] https://www.docker.com/

The Lekta application is compiled and deployed with a REST API wrapper. It is configured, using environment variables, with the URL of the back-end server in order to perform back-office operations. The API defines a simple interface to open new dialogues, send user input, and receive the generated answer (see appendix A.4).

The back-end server is a self-contained system for the communication between semantic queries and databases. It is responsible for providing the GraphQL server, the database translator, and database connectivity layer. The input communication channel is defined by the GraphQL schema, in the form of semantic queries. The database translation is intended to be independent of the language, the database structure and underlying technology. It only handles semantic queries and returns database results.

The container with the server should be self-sufficient in terms of connectivity. Inside it, incorporates the necessary plug-ins to connect to the different database technologies. Using ODBC technology, it ships with plug-ins for connecting with MySQL and PostgreSQL databases. Others may be added in the future. The `Dockerfile` for this image constructs a system with all the necessary components to accomplish this (see appendix A.1). Adding more capacities can be done in the same image definition.

The third container includes the database itself. Preparing a new database to work with the other containers is quite easy. The system have been designed to ease this process. In order to deploy a new database and use it, first the user writes a manifest file with some basic information about the database, such as the technology in use, main table and so on. Second, the database schema and initial contents go to a dedicated folder, that will be loaded at runtime. Finally, the system needs to know about the database to use, for this an `.env` file is written with the database name and other connectivity information. The orchestration system will know what database to deploy and the back-end server which database to connect and how. For more details about the configuration see the appendix A.1.

Container Orchestration

All the containers can be deployed at once using a container orchestration system. In the design of this system, the technology chosen to perform this task is docker-compose[8]. With this system in place, all you need to do to run all the components is typing the following command:

```
$ docker-compose up -d
```

But this technology not only makes it easier to manage multiple containers. Because all the different components are containerized in their corresponding images, one can reproduce them in as many containers as needed. This allows the deployment for high-demand environments. Just taking a step further, with the automatization of scaling, the system could be deployed globally across clusters of hosts.

The configuration is done with the `docker-compose.yaml` file. See appendix A.1) for more details.

[8]https://docs.docker.com/compose/

Chapter 6

Use Cases

In this chapter we are going to try out our prototype using the experimental environment explained in section 7.2, putting it to work in different running conditions. During the process, we will get the opportunity to explain the inner functioning, how the system reacts to user input and the processing that takes place until the final answer is generated.

For this exposition, we will make use of a tool present in the Lekta ecosystem called `synclekta` [29]. This tool is a read-eval-print loop for dialogue systems, built specifically for this technology. Several traces can be activated in order to understand what is happening behind the scenes.

6.1 Basic Case: Full Pipeline

Let's first see how the system responds to a simple user query. This will serve the purpose of introducing the full pipeline, with all the processing steps in detail. We start with a user request:

> *User*: Find how many goals were scored during the Euro cup 2016

Input Query Interpretation

The first module that comes into action is the interpreter. This module makes use of the Lekta NLP pipeline. This pipeline starts with the tokeniser. It works based on the types defined plus the lexicon available. The complete lexicon is compiled from both the framework and the database-specific terms defined by the user (see section 5.2.1). The output trace looks like this:

```
UserProference:
    Find how many goals were scored during the Euro cup 2016

Tokenizer:
* Term: [find]              0- 1 > Category: : lexVerb
* Term: [how many]          1- 2 > Category: : lexDBFunction
* Term: [goals]             2- 3 > Category: : lexDBTarget
* Term: [during the euro cup 2016]
                            5- 6 > Category: : lexDBTopic
```

```
* Term: [were]              3- 4 <NonExistentTerm>
* Term: [scored]            4- 5 <NonExistentTerm>
```

This indicates that four lexical terms have been found in the lexicon. The first one, ‘find’, is a verb captured by the dialogue framework. The second, ‘how many’, comes from the NLIDB framework and refers to an information function. The other two are defined at the project level. ‘goals’ is a target in the database (a table) and ‘during the euro cup 2016’ is a database topic. The rest is ignored.

It is interesting to note that this chunk-based lexicon will produce the similar results for questions like ‘find how many goals’ (keyword-like question) or ‘find how many, er..., you know, goals were scored’ (interrupted phrase), among others. One of the advantages of this approach is its robustness to this kind of noise. We will go into detail about this type of phenomena in the next section.

After the tokeniser, the parser enters the scene. It will work based on the rules defined in the dialogue framework, as well as the NLIDB framework, and, finally, on project-level rules, if any. For more details regarding the entities that will come into play in this phase, the reader can review section 5.2.2. This is the output of the parser:

```
Parser:
* Vertice:  [17] [0-1]
     ProferenceChunk
       |> TaskDialogueAct
            |> ActionDomain
                 |> Verb
                      |> lexVerb (find)
* Vertice:  [14] [1-2]
     ProferenceChunk
       |> Parameter
            |> DBFunction
                 |> lexDBFunction (how many)
* Vertice:  [16] [2-3]
     ProferenceChunk
       |> Parameter
            |> DBTarget
                 |> lexDBTarget (goals)
```

Here we can see the parse-tree generated by the Lekta parser. At the top of the tree, we find the chunks detected by the lexical chunk-grammar. This first round of parsing is done by one of the simple lexical rules we have explained in section 5.2.2. At this point, the parser has risen two `Parameters` for the semantic grammar based on these chunks, apart from the verb captured by the dialogue framework.

The elevation of content for these types is carried out by the unifier. This implements a unification algorithm that checks the appropriateness of content structure from the RHS terms of the rules to the LHS ones. The result looks like this:

```
Unifier:
* Lkton:  [CutPoints:[17] 0 - 1] ProferenceChunk
   > ProferenceChunk:(TaskDialogueAct:(Action:{(ActionDomain:(
```

```
        ActionRoot      :'find',
        ActionDomainName:'LOCATE',
        ActionTense     :'present',
        ActionNegated   :False))}))

* Lkton:  [CutPoints:[14] 1 - 2] ProferenceChunk
   > ProferenceChunk:(Parameter:(
        ParameterCategory:'terminal',
        ParameterType    :'DB_FUNCTION',
        ParameterValue   :(DBFunction:(FunctionName:'COUNT'))))

* Lkton:  [CutPoints:[16] 2 - 3] ProferenceChunk
   > ProferenceChunk:(Parameter:(
        ParameterCategory:'terminal',
        ParameterType    :'DB_TARGET',
        ParameterValue   :(DBTarget:(
            TargetPath: goal_details.goal_id'))))
```

Here we can see the contents of the verb and the two parameters. These contents define the semantics of the chunks detected. In the case of parameters, we have the parameter type, and the data associated with it. For this input phrase, the contents are the function name for the function parameter, and the path for the target parameter.

After this step, we have at our disposal three chunks containing the semantics of the corresponding utterances. We have abandoned the lexical domain and can operate over semantic elements alone. What we are looking for, now, are patters of these semantic pieces. We have a verb in the domain of searching, an information function and a database target. This is a pattern that indicates the intention of querying information. To detect this pattern, in our semantic grammar, we have a rule that searches for this particular sequence of chunks. We have called it `PragmaticMapper_LocateFunctionTarget` in the NLIDB framework. We have explained these rules in more detail in section 5.2.3. The semantic production rule says something like this in pseudo-code:

```
Chunk:Action/locate
Chunk:Parameter/DB-function
Chunk:Parameter/DB-target
->
DAct:Request/Action/NLIDB
DAct:Inform/Parameter-provide/target {
    ^.Function = #2  // DB-function
    ^.Argument = #3  // DB-target
}
```

From the three consecutive chunks we have mentioned before, we produce two dialogue acts. The first one is the intention of querying, which is expressed as an action request for a natural language query. The second one is the query content, given as a parameter to be used in the NLIDB skill. Here we construct a target, in the sense of a Semantic Query target, not to be confused with what we call a target at the lexicon level and the chunk level. This object is compound by a function and an argument, and their respective contents come from the semantic level parameters (function name and argument

path).

Query Translation

In the dialogue manager, the presence of the first dialogue act in the pipeline, activates the skill in charge of managing queries, which we have called NLIDB. As a minimum for asking the back-office, this skill requires that a semantic query target is set at least. In other terms, it requires that, when we are looking for something, that something is defined. When this happens, like it does in our case, it transforms the skill parameters into a JSON object according to the format defined in the GraphQL schema (see section 5.3.1). Finally, the back-office function is called with the following payload:


```
{
    "operationName": "nlidb",
    "query": "query nlidb($semQuery: SemQuery!) { nlidb(semQuery
       : $semQuery) }",
    "variables": {
        "semQuery": {
            "command": "SELECT",
            "targets": [{
                "argument": "goal_details.goal_id",
                "function": "COUNT"
            }],
            "conditions": []
        }
    }
}
```


This payload arrives to the back-office server, thanks to the intermediate operation of the back-office plug-in, which performs the HTTP POST request to the server. *Scotty*, the web server, processes the request and passes it onto *Morpheus*, the GraphQL infrastructure. This back-end first validates the format of the `semQuery`, which is correct, and transforms the contents into the corresponding Haskell type `SemQuery`. Then it enters the resolution method which, in turn, calls the `toSql` function over the semantic query. For further reference about these components, please go to section 5.3.1.

The instanced method `toSql` of `SemQuery` Haskell type, first looks to transform targets into selection columns. There is only one target to translate, with both function and argument, which produces the following SQL code: `COUNT(goal_id)`. Now it is the turn of the `resolveRange` function, which only finds a table. Therefore, this is translated into the from-statement `FROM goal_details`. These functions have been described in section 5.3.2. There are no conditions, therefore the final query looks like this:

```
SELECT COUNT(goal_id) FROM goal_details
```

The resolver handles this query to the database back-end (see section 5.3.3). Thanks to the given configuration, the `query` function is able to produce an ODBC connection

string adequate to this database. And now, it can use the HDBC function `quickQuery` to pass the database our SQL code.

The database response, of type `[[SqlValue]]`, is now serialized to conform to the GraphQL operation response type. Thus, the resolution of the GraphQL operation has been successful. The GraphQL framework can now handle the data to the web server, which finally can return a 200 HTTP response with the results in the body:

```
{
    "data": {
        "nlidb": "[[108]]"
    }
}
```

Response Generation

Now, we are back at the NLIDB skill in the Lekta application, and we have the answer from the back-office server. It is the turn of the response generator (explained in more detail in section 5.4). The first thing this module does with the data received, is parsing the string and turning it into a Lekta JSON object. Immediately after, the results are classified. After removing the table wrappers, we end up with a single result.

At this point, the system generates an output dialogue act of type `Inform/Statement/db_result_value`, that indicates the intention of giving an answer to the user, with certain format. At the same time, it associates with it the information to be presented. That is, a JSON value with the results without wrappers.

Finally, we arrive at the generation phase in the Lekta pipeline. We have defined a generation rule that is triggered when it encounters a dialogue act of the type mentioned above. Because the result type is a single value, the system knows it can output it as the main answer in the written (or spoken) channel. At last, filling in a template with the serialized version of the value, the ultimate answer can be shown to the user:

System: The answer is: 108.

6.2 Linguistic Phenomena

Extra-grammatical Sentences

We have mentioned, while we explained the processing pipeline, that the system was resilient to some types of noise. We can now see these phenomena in more detail. Let's consider a variant of the phrase we started with: 'Find hw many gaols were scourd during the Euro 2016'. Let's check the tokeniser trace to see how it is processed.

```
Tokenizer:
* Term: [find]        0- 1 > Category: : lexVerb
* Term: [hw many]     1- 2 > Category: : lexDBFunction
     -> : [how many]  By Insertion/Pos:1/Symbol:[o]
* Term: [gaols]       2- 3 > Category: : lexDBTarget
```

```
     -> : [goals]  By Reversal/Pos:1/Symbol:[a]
* Term: [the]        6- 7 > Category: : lexDetThe
* Term: [euro 2016]  7- 8 > Category: : lexDBTopic
* Term: [were]       3- 4 <NonExistentTerm>
* Term: [scourd]     4- 5 <NonExistentTerm>
* Term: [during]     5- 6 <NonExistentTerm>
```

Here we can see Lekta spell corrector in action. ‘hw many’ has been corrected to ‘how many’, and ‘gaols’ to ‘goals’. In the trace we also see the correcting operations performed. These terms can be classified now as `lexDBFunction` and `lexDBTarget` respectively. Therefore, these types are parsed resulting in the same tree as in the basic case. The pipeline will look exactly the same as before, resulting in the correct answer ‘108’.

Activation by Parameters

Consider the query: ‘total goals’. This is the typical keyword-like question a user may ask when he wants to formulate a simple query. And this is what the tokeniser generates:

```
Tokenizer:
* Term: [total]  0- 1 > Category: : lexDBFunction
* Term: [goals]  1- 2 > Category: : lexDBTarget
```

Again, we have two terms that are classified as an information function and a database target. They will be parsed in a similar way to the basic case, but here we end up with a simpler tree, with only two chunks as vertices. Surprisingly, we arrive to the same answer.

At first sight, there seems to be missing information. In the first place, there is no information about the topic we are talking about. We are assuming the default theme, and the database to query, is the one we have defined in the experimental environment. But, in a multi-database environment, we could have found this topic previously in the conversation, and use it as context.

We can also see here the chunk grammar in action. Because we have defined the meaning of these two chunks, and the pattern they form is meaningful as well, we have all we need to form a query. The rest of the information is superficial at this level. This allows the formulation of keyword-like questions, as the one we are considering.

But still, we could activate the skill without information of the action to perform. That is, we are missing words like ‘find’ or ‘tell me’, that indicate we are asking for information. This is what the chunk with the locate action verb (`lexVerb`) did in the basic case, what is usually called an (explicit) intent. But the pattern is enough for us to know we are before a query, so we just need to implement this in a rule. Indeed, we use a different rule to generate the DActs that activate the skill. This time the rule is like this in pseudo-code:

```
Chunk:Parameter/DB-function
Chunk:Parameter/DB-target
->
DAct:Request/Action/NLIDB  // activates the skill
DAct:Inform/Parameter-provide/target {
```

```
    ^.Function = #1  // DB-function
    ^.Argument = #2  // DB-target
}
```

Sentence Noise

The chunk grammar is also useful to make the system resilient to noise. Check the parsing tree of a sentence like: ‘how many... what do you call’em... venues there were in the tournament’.

```
Parser:
* Vertice:  [33] [0-1]
     ProferenceChunk
        |> Parameter
            |> DBFunction
                 |> lexDBFunction (how many)
* Vertice:  [35] [12-13]
     ProferenceChunk
        |> Parameter
            |> DBTarget
                 |> lexDBTarget (venues)
* Vertice:  [37] [13-14]
     ProferenceChunk
       |> Parameter
            |> Number
                 |> lexNumberValue (there)
```

We have reduced the parse-tree to three vertices. Because a lot of the sentence is meaningless to us, we can simply ignore it and get by with the chunks detected. What is meaningful to our task is coded in the lexicon chunks, and nothing else.

We have ended up with the same pattern as the basic case, except for a third chunk. The spell corrector has transformed this word to the number ‘three’ and classified it as a number. We can recover from this mistake when we get to the second level of parsing. Because, there is no pattern that will consider a number at that position, the function-followed-by-target rule is activated. The rest of the pipeline is very similar to the basic case. We finally arrive to the correct answer ‘10’.

NLIDB as a Skill

One of the advantages of having developed the NLIDB as a component inside a dialogue framework, is that it can live together with other functionality already present in the framework. Consider this short dialogue:

> *User*: Tell me how many countries participated in the Euro 2016 and, by the way, what time is it?
> *System*: The answer is: 24. It’s 12:49 PM.

The first part of user’s turn looks like one of the sentences we have analysed before. But the second part does not. We actually don’t have any time information in the database.

The dialogue framework is able to distinguish two different actions that are performed by dedicated skills. The first one is processed by our NLIDB, the second one by a time-telling skill present by default in the framework.

6.3 Complex Queries

Filters

Let's now consider phrases that produce more complex queries and check how the system can handle them. We can start with explicit filters that appear in sentences like 'find the total number of matches that ended in victory'. As usual, the pipeline starts with the tokeniser. Let's see what is new here:

```
* Term: [that]          4- 5 > Category: : lexDetThat
* Term: [in]            6- 7 > Category: : lexPrepIn
* Term: [victory]       7- 8 > Category: : lexDBValue
* Term: [ended]         5- 6 <NonExistentTerm>
```

And in the parser:

```
* Vertice:  [23] [6-8]
     ProferenceChunk
       |> Parameter
            |> DBCondition
                 |> lexPrepIn (in)
                 |> lexDBValue (victory)
```

The expression 'in victory' has been transformed into a database condition. We expect expressions with the pattern 'in ...' to introduce a condition. If, what follows, is a value, we can turn it into an equality condition. We can see this in the unifier:

```
* Lkton:  [CutPoints:[23] 6 - 8] ProferenceChunk
   > ProferenceChunk:(Parameter:(ParameterCategory:'terminal',
         ParameterType  :'DB_CONDITION',
         ParameterValue :(
             DBCondition:(SQLValue  :''WIN'',
             DBOperator:(OperatorName:'EQ'),
             DBTarget  :(TargetPath:'match_mast.results')))))
```

This condition will be translated into an element in the `conditions` field for the `nlidb` operation.

```
"conditions":[{
    "subject":"match_mast.results",
    "operator":"EQ",
    "value":"'WIN'"
}]
```

In the back-end, following the implementation of the `toSql` function, the operation argument is translated into the final, and expected, query:

```
SELECT COUNT(*) FROM match_mast WHERE match_mast.results='WIN'
```

Implicit Filters

A similar case to the previous arises with the use of database values alone, in expressions that doesn't indicate a condition. This happens in phrases like 'count total victories'. Here, the term 'victory' takes the form of a target. It is the thing that we want to count, but there is no such element in the database. That is, there is no `victories` table, for instance. Instead, the implementer have decided, for example, to add a classification column called `results`, that contain a particular value for victories, ie `'WIN'`, in the match results table.

In such cases, at the semantic level, we look for a function followed by a database value, and turn it into a target plus a condition. The rule that looks like this:

```
Chunk:Parameter/DB-function
Chunk:Parameter/DB-value
->
DAct:Request/Action/NLIDB
DAct:Inform/Parameter-provide/target {
     ^.Function = #2          // DB-function
     ^.Argument = #3.Target  // DB-value target
}
DAct:Inform/Parameter-provide/target {
     ^.Target = #3.Target    // DB-value target
     ^.Condition = 'EQ'
     ^.Value = #3.Value       // Db-value SQL value
}
```

From there, the pipeline will follow a path similar to the previous case. We end producing a slightly different query, that gives the same answer:

```
SELECT COUNT(results) FROM match_mast
WHERE match_mast.results='WIN'
```

Joins

Consider a request like 'tell me the stadium of the final match'. The first part of the pipeline will produce a similar analysis as a case with conditions. In particular, the parser will find a target ('stadium') and a condition ('of the final match'). The difference in this case is that these two elements are located in different tables. We can make this explicit seeing the resulting operation payload:

```
"targets": [{
     "function": "",
     "argument": "soccer_venue.venue_name"
```

```
}],
"conditions": [{
    "subject": "match_mast.play_stage",
    "operator": "EQ",
    "value": "'F'"
}]
```

So we have two elements in tables `soccer_venue` and `match_mast` respectively. For what the Lekta application is concerned, this is a perfectly valid linguistic analysis of the elements of the query. We don't look for consistency, in terms of relatedness, of the data at this level. This is passed on to the back-end server, which will try to pair the two tables.

In the `toSql` function, everything works as usual until it arrives to the `resolveJoins` method (see section 5.3.2). This function receives as second argument the list of tables of targets and conditions, ie `["soccer_venue", "match_mast"]`. The first argument is the relations graph. This graph is constructed querying the database for the pairs foreign key-primary key. Those pairs constitute the edges of the graph. So we try to find a relation between the two tables. This specific task is performed by the function `findRelation`. Traversing the graph, it finds a direct relation between those two tables in the pair `("match_mast.venue_id", "soccer_venue.venue_id")`.

The rest is constructing the SQL join expression for the from-clause. The query the back-end sends to the database is the following:

```sql
SELECT venue_name FROM soccer_venue
    JOIN match_mast ON match_mast.venue_id=soccer_venue.venue_id
WHERE match_mast.play_stage='F'
```

This is the expected query we should generate. Finally, the answer can be returned to the user:

System: The answer is: "Stade de France".

Chapter 7

Evaluation

It is important to assess the quality of a system in order to, both, measure progress in successive versions of the implementation and to compare the system against others of similar characteristics. In the Natural Language Processing domain, this is a particularly difficult task that, many times, involves subjective criteria. There is plenty of literature about this topic, but we are not going to delve too deep into the problem of evaluation of natural language processing systems.

In this prototype, we are going to focus on the ability to set milestones of functionality that will allow us to ascertain progress. This same evaluation framework permits the comparison with other systems (as is done in [2]). But this methodology will require the development of new environments, that are not ready at the time of writing. Instead, we are going to base our evaluation on the performance over a specific database setup. The main performance indicator will be the system's ability, or not, to answer user questions correctly, giving the expected database results.

7.1 Evaluation Framework

Lekta technology provides a mechanism for evaluating dialogues called *DialogueActivity* [29]. This is a dedicated information channel that developers can use to emit meaningful information during the execution of the system. The technology defines functions that can be used to emit a message, or tag, with coded information at certain point in the dialogue flow.

Along the turns in a dialogue, for example, the processing of the user input can activate certain rule. We can then place a checkpoint in this rule to emit a *DialogueActivity* message to indicate this activation. Or we can emit a message with the generation of a dialogue act, with its particular taxonomy. With many of these checkpoints, we can evaluate a functioning dialogue as a predictable list of messages of this kind. The same user input phrases must always generate the same *DialogueActivity* stream of tags. Note that this type of evaluation is only possible when the system infrastructure is based on a symbolic approach. A system entirely based on statistical methods can not employ this methodology because we can't control the intermediate stages of the processing and take representations.

In this implementation, the system produces *DialogueActivity* tags after calling the back-office service, with the type of results obtained and its data. Doing this, we can

evaluate the actual result the system produces in response to a user question against the expected answer. The format of these tags is as follows:

```
|INFO_ITEM/NAME:db_result_value/TYPE:<result_type>/VALUE:<data>|
```

Where `<result_type>` classifies the type of answer as is done in 5.4. Whereas `<data>` is the actual information returned by the database, but cleaned of unnecessary wrappers.

In order to use the messages emitted through this channel, the Lekta ecosystem includes a dedicated tool called *ktUnit*. With this tool we can define dialogues, that will be evaluated against the expected list of *DialogueActivity* tags. The configuration is done in a file with a special format described in [29], which is passed as argument to this tool. We can also classify the tests in three levels of granularity, so we can run them more selectively. All the dialogues defined this way are automatically evaluated in batch and the results shown in the screen.

In the listing in appendix A.3, the reader can find a complete file containing one of these tests. The syntax used in this file differs slightly from the standard, thanks to the use of preprocessor constants and macros. The format of such tests is always the same, this way we can reduce the boilerplate needed for simple tests. The test itself looks like this:

```
&KTUNIT_TEST_SIMPLE
(
  "|INFO_ITEM/NAME:db_result_value/TYPE:SINGLE_VALUE/VALUE:24|",
  find the number of countries that participated
  in the EURO cup 2016
)
```

In this test, for instance, we are entering the system the following phrase: ‘find the number of countries that participated in the EURO cup 2016’. After processing the query, the system must produce an information item containing a single value with the value ‘24’, which is the correct answer to this question. We codify the expected answer as a *DialogueActivity* tag with the format mentioned before. This tag is the first argument we pass to the macro in the example above.

If we run this test using `ktunit` command, we would obtain the following results. Note that the caret character means success. We also check the return state of the command, which give us the total of unsuccessful tests.

```
$ ktunit -SG DBEuro2016 -SU Query/Simple -SL ENGLISH/02 Test.ktu
$> ktunit Target SchemeGroup:DBEuro2016 - SchemeUnit:Query/
   Simple SchemeLabel:ENGLISH/02
$>       ^

$ echo $?
0
```

This tool works perfectly for testing dialogues, but it doesn’t integrate well in an automated test-driven development setup. This is so because, if we place a test that we know will fail, the final result of the whole test suite will result in a failure. We would

like to mark those tests that do not pass yet as to-do tests. Doing so, the battery of tests won't fail because of know errors. With this method, we can use the tests as a harness for regression errors, while we work on new functionality.

To accomplish the former, a test wrapper for ktUnit has been developed. Remember that `ktunit` command returns the number of errors and that we can run tests selectively. We can develop a function that will run an individual test, then evaluate it and generate some output that can be interpreted by a test harness. The implementation of this function, called `ok_ktunit`, can be found in the appendix A.3. The arguments of this function are simply the classification levels of the test. A single test would now look like this:

```
ok_ktunit DBEuro2016 Query/Simple ENGLISH/01
```

The test output format of choice is TAP [32]. This is a technology agnostic testing format, that just requires that the system can produce output in the specified format. It is a widely used standard, for which many tools are available.

We can now run the whole set of tests using a bash script. The reader can find an example of such file in appendix A.3. This script in turn can be handled by a TAP harness such as `prove`. This is a standard tool that ships in all mainstream Linux distributions. The output of the execution of the harness would look something like this:

```
$ prove Skills/DBEuro2016/Tests/test.sh
Skills/DBEuro2016/Tests/test.sh .. 1/65
...
Skills/DBEuro2016/Tests/test.sh .. 64/65
Skills/DBEuro2016/Tests/test.sh .. ok
All tests successful.
Files=1, Tests=65, 14 wallclock secs ( 0.07 usr  0.00 sys +
   7.97 cusr  4.31 csys = 12.35 CPU)
Result: PASS
```

7.2 Experimental Setup

For the evaluation of this project we have chosen a sample database created by the staff of an online web technologies academy called w3resource[1]. The database of choice is a soccer database containing statistics about the Euro 2016 cup tournament [35]. The reader can find the full schema in the appendix A.1.

The rationale behind this decision is, first, to use a rather realistic case or, said the other way around, not to use a database specifically designed to be used with a natural language interface. This is interesting because we want to test the flexibility of the methodology to adapt to new environments. We are interested in a database designed for any purpose, and try to adapt the system to it.

This database in particular poses some challenges to build a natural language interface using our model. For instance, there are entities which have different column names

[1] https://www.w3resource.com/

in different tables. This creates an ambiguity, making it difficult to point to them in a denotational model as the one we propose.

There isn't a perfect matching between world objects and database representation. For example, it uses classification columns, which is a common implementation technique. This makes it difficult to translate the relation between the object and the classification. In the other hand, some features of this database ease the process of building an NLI, like being normalized and defining the primary and foreign keys.

The second reason for this choice is that this course contains an incremental set of natural language questions that the students are meant to translate into database queries. We can take advantage of this program and use it for an incremental implementation of the system. We can transform the natural language queries and the answers into a test suite for setting a test-driven development environment. This is what we have done for a selection of queries with their answers, using the methodology explained in the previous section.

In this setup, the SQL student is the NLIDB system, rather than an IT student at some university. The set of problems the teachers have proposed as exercises, constitute our TAP harness. We can now start implementing functionality and mark to-do tests as done when the system is able to answer a question.

7.3 Results

The tests in this course are divided in three groups: simple queries, join queries, subqueries. Let's see the performance of the prototype in this setup, after evaluating it with the framework explained before.

The prototype proposed, using a chunk-based grammar, and about 150 lexicon entries, is capable of answering several questions. This is a summary of the results:

- Simple queries:
  - Can answer the majority of simple questions.
  - Can do so even in noisy conditions, like with typographic errors and grammatically incorrect questions.
  - It handles conditions that hasn't got a complicated structure.
  - Can use some concepts, when they can be expressed in the lexicon.
- Queries with joins:
  - It can perform joins with simple join paths.
  - The exercises in this category mix joins with subqueries and thus depend on the performance over them. It is difficult to measure this feature alone with this setup.
- Subqueries:
  - This type of grammar is not well suited to handle subqueries.
  - Some subqueries can be avoided with the use of concepts.

Chapter 8

Conclusions and future work

The architecture chosen in the design has proven to be adequate to implement an NLIDB system. The main advantage of this design is its modularity. With the definition of interfaces between the different functional modules, we ensure their interoperability while being flexible in terms of implementation. Doing this, we are also following the design principle of Separation of Concerns. In addition, we are using the technologies that are more suited to the problem at hand. We use Lekta for language processing tasks, and a functional language, Haskell in this case, when we are dealing with data transformation alone.

Respecting the interfaces, this design admits several insertion points that allow the substitution of modules or implementations without changing the overall operation of the system. In the Lekta application, rule-based development permits the addition of new rules without changing the existing ones and without redefining the program flow. In the back-end, function signatures define a functional interface that, when implementations abide to it, the processing flow is preserved. For this, Haskell's clarity and purely-functional design is of great help. Also, GraphQL guarantees the interoperability with any web-based back-end that implements the schema we have fixed.

A chunk-based grammar, like the one we have defined, has shown to be appropriate to cope with, at least, a subset of the language used in querying information. While the linguistic coverage it can deal with is limited, it is a resilient strategy to handle simple queries, that tend to be keyword-like sentences. Sometimes with grammatical errors or other extra-grammatical utterances.

The automated evaluation system has proven to be a vital tool in the development of the NLIDB and can be so, by extension, to any NLP program. The development of NLP applications is bound to produce regression errors. Due to the interconnectedness of language, a change that affects a linguistic element will often produce undesired effects that break other functionality. Having a testing harness in place, with well-defined objectives, or expected behaviour, the development of features is much more reliable. This same system can be useful to define some performance checkpoints that can be used to measure the quality of the system. Or set milestones that indicate progress, as we have done with our testing environment. At the same time, if a proper comparison methodology is defined, this system can automate the comparison with systems of the same category.

Project organization is another interesting design feature to highlight here. The development of the database-specific code has been confined to the development of a user-level

skill with only lexicon. This poses an interesting step towards easing the portability and configuration of this type of systems. Code organization in a domain-independent framework and database-specific skills has been one of the elements that have facilitated this level of customization. At a pragmatic level, the micro-services structure, and the containerization of services using Docker, makes this a system easy to deploy and capable to scale to the level necessary to serve in high-demand environments.

Regarding the improvements, at the level of components, this type of system can take advantage of, we can mention here the most important or immediate. First, regarding the understanding phase of the pipeline, a more powerful grammar would be necessary to analyse more complex queries, with more intricate syntax, that express more detailed requests. It would implement a dependency grammar that looks into the syntactic structures people use when requesting information. This requires research on the linguistic side as well as proper implementation. This kind of grammar can be plugged in the system to inject dialogue acts to the dialogue manager, so it can live together with the chunk grammar. In order for the two to collaborate, the dialogue manager would have to implement some orchestration mechanism with some heuristics.

In the back-end service, some obvious improvements we can account for would be a better join path resolution and proper error handling. The join path algorithm can be more sophisticated and take into account other sources of information in order to resolve successfully. It could, for instance, implement type checking in order to solve some ambiguities. The database provides useful information when an error is produced. Database systems normally make a good diagnosis of the problems they encounter and return them properly classified. If we can take advantage of this information, some errors can be recovered, or the service can send them back for further analysis or to improve the communication with the user. When it comes to the generation module, we can think of better classifications in order to present the results in the proper format. If a better grammar is in place, we can also think of making the answer more natural, by preserving correspondence in time, person, etc.

Another important point of improvement is the dialogue manager. Taking into account that we have built our system using a dialogue framework, we could use its features more cleverly. This type of system can help a lot in the communication with the user, in order to clarify the questions. We have mentioned before the errors generated by the database, but any type of error, inconsistency or information gap is an opportunity to ask the user for clarification. Also, using the dialogue framework we can assist the user to understand what capacities the system has, both linguistic and functional. Finally, at the linguistic level, it can be useful in order to resolve anaphoric expressions and ellipsis, using the dialogue history.

Perhaps even more interesting, regarding the dialogic features of this architecture, is exploring the possibility of focussing on the dialogue system, making the NLIDB its subordinate. This opens an interesting area of research, where we can invert the terms and, instead of thinking of one-shot questions with answers, like in an NLIDB system, we can think how to use this type of system to access different sources of information and improve the dialogue system. Having this data access capacities would be a very interesting feature in a dialogue manager.

If we consider the problem at hand with a wider perspective, what we should question ourselves first is the appropriateness of the denotational model for semantics. That

is, the model that have been used to implement the types and lexicon. Definitely, the model should be expanded to cover more cases. But, more generally, this schema works properly for nominal expressions, but it falls quite short when it comes to make relations explicit. With the development of more complex grammars, we can relay on them to model relations in language and keep denotational semantics for nouns. Or we can think of more elaborate world models. The denotation itself could be improved to point to entities referred using other representations. That is, we could point to entities implemented in other technologies, like SPARQL, XPath or many others.

Regarding the underlying database systems, we can question the suitability of the relational model in order to represent certain information. This schema has its own expressive limitations. If we try to take the relations or functions in a RDBMS as relations or functions of objects in the world, and translate them literally, we will only come close to express many states of affairs. What is appropriate in terms of storage, may not be good to convey the language that expresses it. We should keep this in mind when adapting a linguistic interface to a database. Specially when we are dealing with real-world databases, where we can not control the design of the schema. This circumstance is made clear by the fact that any property can be implemented in a database in infinite ways, all functionally equivalent.

At any rate, we should study other approaches to semantics, test their performance, and try combinations of them. For instance, it would be interesting to mix this approach with statistical techniques, such as named entity recognition or n-grams. That is to say, combine symbolic approaches to semantics with models closer to meaning-as-use. It would be beneficial to expand the scope of the problem and open the research to combinations of symbolic and empirical approaches.

The other model we should question is the semantic query. The focus of this prototype was more on the architecture, and thus this model has limitations in order to express complex questions. These shortcomings should be addressed. This model could be expanded by redefining the GraphQL schema, to make it more expressive. But, going further, we should aim for a more general information retrieval model, compare the different approaches and study the problem in its generality. We should try to answer the question: what is the semantic content of the act of querying? What could a model that contain all the elements present when we ask for information, and that preserves its structure.

Appendix A

A.1 Containers

Container Orchestration

Environment variables

LEKTA_PORT: Port of the Lekta application REST API.

BO_PORT: Port of the back-office server.

DB: The name of the database. This defines both the database to deploy and the database the back-end server will use by default.

DB_TYPE: The database technology. This defines what Docker image should be deployed. It also tells the back-end server how to handle the database connection.

DB_PORT: Database connection port.

DB_USER: Database connection user name.

DB_PASS: Database connection user password.

Docker-compose Definition

```
version: "3"
services:
  api:
    build:
      context: ./
      dockerfile: ./docker/api/Dockerfile
    ports:
      - ${API_PORT:-3000}:3000
    networks:
      - net

  db:
    image: ${DB_TYPE}:latest
    ports:
      - ${DB_PORT:-4000}:${DB_INNER_PORT}
```

```
16     environment:
17       - MYSQL_ROOT_PASSWORD=${DB_PASS:-1234}
18       - POSTGRES_PASSWORD=${DB_PASS:-1234}
19     volumes:
20       - ./docker/db/${DB}/initdb:/docker-entrypoint-initdb.d
21     networks:
22       - net
23
24 networks:
25   net:
```

Listing A.1: docker-compose.yaml

Back-end Server Container

Back-end Server Dockerfile

```
1  FROM ubuntu:19.04
2
3  RUN apt-get update && apt-get install --assume-yes wget unixodbc
      -dev odbc-postgresql
4  RUN cd /tmp && \
5      wget https://dev.mysql.com/get/Downloads/Connector-ODBC/8.0/
          mysql-connector-odbc-8.0.16-linux-ubuntu19.04-x86-64bit.
          tar.gz && \
6      tar -xzf mysql-connector-odbc-8.0.16-linux-ubuntu19.04-x86
          -64bit.tar.gz && \
7      cp mysql-connector-odbc-8.0.16-linux-ubuntu19.04-x86-64bit/
          lib/* /usr/lib/x86_64-linux-gnu/odbc/ && \
8      rm -rf /tmp/*
9
10 COPY docker/api/odbcinst.ini /etc/odbcinst.ini
11
12 COPY app/ /opt/app/app/
13 COPY .stack-work/install/x86_64-linux-tinfo6/lts-13.28/8.6.5/bin
      /nlidb-bo-exe /opt/app/server
14 COPY .env /opt/app/.env
15 WORKDIR /opt/app
16
17 ENTRYPOINT ["/opt/app/server"]
```

Listing A.2: Dockerfile

Back-end server ODBC configuration

```
1 [postgres]
2 Description=PostgreSQL ODBC driver (Unicode version)
3 Driver=psqlodbcw.so
4 Setup=libodbcpsqlS.so
5 Debug=0
6 CommLog=1
7 UsageCount=1
```

```
8
9 [mysql]
10 Description=MySQL ODBC driver (Unicode version)
11 Driver=libmyodbc8w.so
12 Setup=libmyodbc8S.so
13 Debug=0
14 CommLog=1
15 UsageCount=1
16
17 [mariadb]
18 Description=MySQL ODBC driver (Unicode version)
19 Driver=libmyodbc8w.so
20 Setup=libmyodbc8S.so
21 Debug=0
22 CommLog=1
23 UsageCount=1
```

Listing A.3: odbcinst.ini

Database Container

Folder organisation

```
1 <database_name>/
2   |- Readme.md
3   |- db.env
4   \- initdb/
5      |- 00-schema.sql
6      \- 01-data.sql.gz
```

Configuration

Example of a `db.env` file for a MariaDB database.

```
1 DB_TYPE=mariadb
2 DB_NAME=main_schema
```

A.2 Database

Relations Script

Query to find the relationships in a PostgreSQL database.

```sql
-- Generate a list of pairs foreign key - primary key.
-- Adapted from: <https://dataedo.com/kb/query/postgresql/list-of-foreign-keys-with-columns>.

select distinct
    kcu.table_schema || '.' || kcu.table_name || '.' || kcu.column_name  as fk,
    rel_kcu.table_schema || '.' || rel_kcu.table_name || '.' || rel_kcu.column_name as pk
from information_schema.table_constraints tco
join information_schema.key_column_usage kcu
    on tco.constraint_schema = kcu.constraint_schema
    and tco.constraint_name = kcu.constraint_name
join information_schema.referential_constraints rco
    on tco.constraint_schema = rco.constraint_schema
    and tco.constraint_name = rco.constraint_name
join information_schema.key_column_usage rel_kcu
    on rco.unique_constraint_schema = rel_kcu.constraint_schema
    and rco.unique_constraint_name = rel_kcu.constraint_name
    and kcu.ordinal_position = rel_kcu.ordinal_position
where tco.constraint_type = 'FOREIGN KEY';
```

Listing A.4: relationships.sql

Experimental Database Schema

See figure A.1.

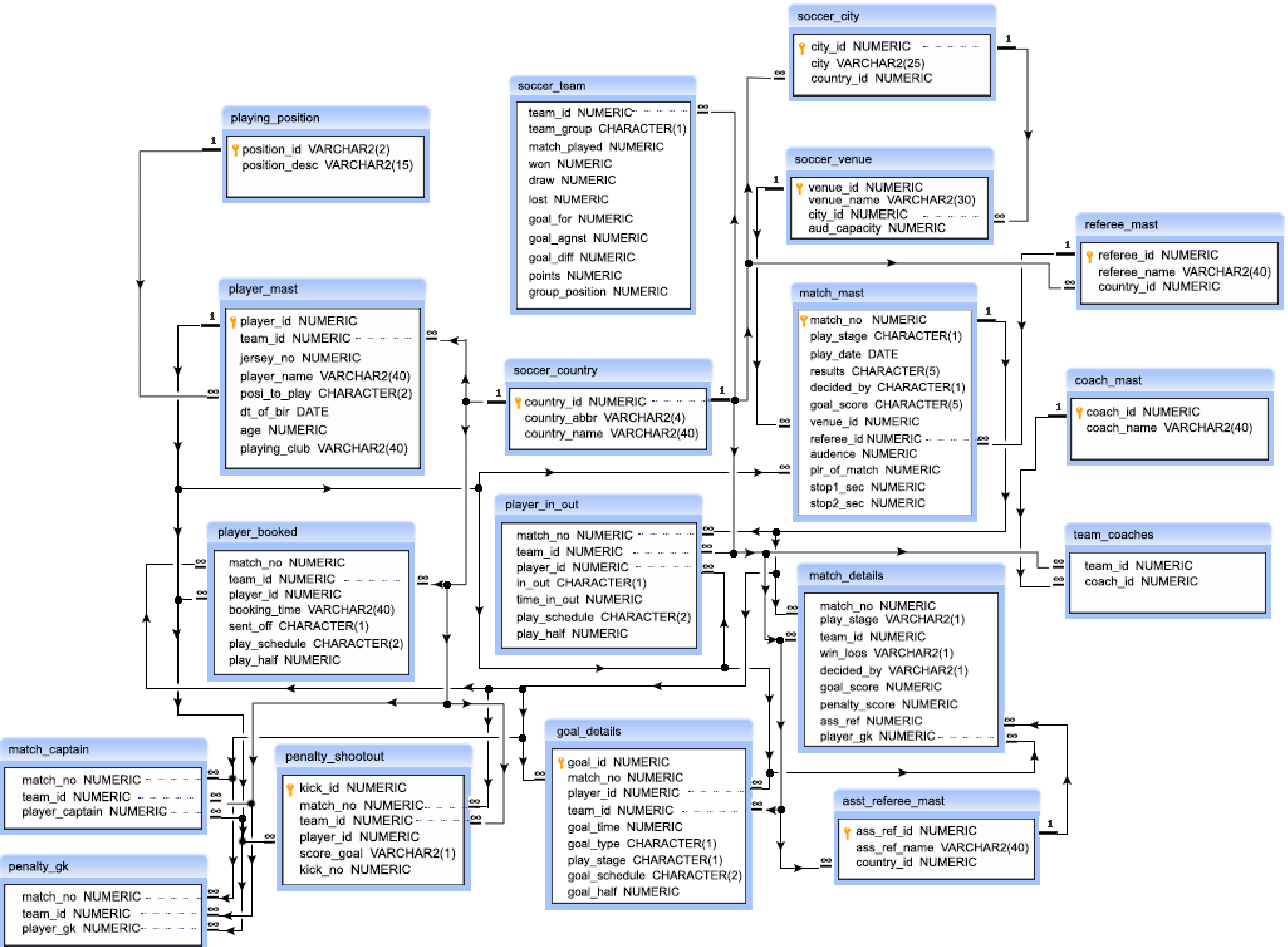

Figure A.1: Euro 2016 database schema.

A.3 Evaluation

ktUnit file

```
#Define SCHEME_GROUP "DBEuro2016"
#Define SCHEME_UNIT  "Query/Simple"

#Define INSTANCE_OPERATION "Fluency"
#Define INSTANCE_LANGUAGE  "English"
#Define INSTANCE_INTERFACE "Written"

#Define SCHEME_LABEL "ENGLISH/02"
#Define TRACE_FILE "Tests/Traces/DBEuro2016-simple-en-02.trace"
#Define STATS_FILE "Tests/Statistics/DBEuro2016-simple-en-02.
   stats"
&KTUNIT_TEST_SIMPLE
(
    "|INFO_ITEM/NAME:db_result_value/TYPE:SINGLE_VALUE/VALUE
       :24|",
    find the number of countries that participated in the EURO
       cup 2016
)
```

Listing A.5: test.ktu

ktUnit harness wrapper function

```
ok_ktunit() {
    local description="$1 $2 $3"
    local directive=""
    if [ "$4" = "TODO" ]; then
        directive=" # $4"
    fi

    if  ktunit -SG "$1" -SU "$2" -SL "$3" "$build_path/Test.ktu"
        > $stdout
    then
        echo "ok - $description$directive"
    else
        echo "not ok - $description$directive"
    fi
}
```

Test script

```
#!/usr/bin/env bash
source Tests/lib.sh  # library with testing functions

echo "1..10"

require db Euro2016
require bo Euro2016

ok_or_bailout_compile DBEuro2016 #1

ok_ktunit DBEuro2016 Query/Simple ENGLISH/01
ok_ktunit DBEuro2016 Query/Simple ENGLISH/02
ok_ktunit DBEuro2016 Query/Simple ENGLISH/03 TODO
ok_ktunit DBEuro2016 Query/Simple ENGLISH/04 #5
ok_ktunit DBEuro2016 Query/Simple ENGLISH/05
ok_ktunit DBEuro2016 Query/Simple ENGLISH/06 TODO
ok_ktunit DBEuro2016 Query/Simple ENGLISH/07
ok_ktunit DBEuro2016 Query/Simple ENGLISH/08
ok_ktunit DBEuro2016 Query/Simple ENGLISH/09 #10
```

Listing A.6: test.sh

A.4 Lekta Application REST API

Environment variables

`API_KEY`: Enabled X-Api-Key authentication

`APP_SRC`: Name of the dialogue source file (*.lkt)

`APP_BIN`: Name of the dialogue binary file (*.olk)

`FLUENCY_VERSION`: Fluency version in use

`LEKTA_THREADS`: Number of dialogue threads allowed by the license

`RESOURCE_MANAGER_URL`: URL of the back-end server

Open dialogue

Opens a new dialogue and expects its ID as response.

Request

```
POST /dialogues
Content-Type: application/json

{
    "language": "English",
    "interface": "Written",
    "operation": "Fluency",
    "context": "{\"valid\": \"json\"}"
}
```

Response

```
200 OK
Content-Type: application/json
Dialogue-Id: 1f48feac-ca39-4af7-8f1b-a4d02fa80530

{
    "id": "1f48feac-ca39-4af7-8f1b-a4d02fa80530",
    "answer": "Hi there! I’m Lekta.",
    "closed": false,
    "context": "{\"valid\": \"json\"}",
    "language": "English",
    "interface": "Written",
    "operation": "Fluency"
}
```

Close dialogue

Closes a dialogue by ID.

Request

```
DELETE /dialogues/:id
```

Response

```
204 No Content
```

Dialogue input

Sends user input to the dialogue given by ID.

Request

```
POST /dialogues/:id
Content-Type: application/json

{
    "input": "Hi Lekta. How are you?",
    "context": "{\"valid\": \"json\"}"
}
```

Response

```
200 OK
Content-Type: application/json

{
    "answer": "I'm fine, thank you.",
    "closed": false,
    "context": "{\"valid\": \"json\"}",
    "language": "English",
    "interface": "Written",
    "operation": "Fluency"
}
```

Bibliography

[1] Steven P. Abney. Parsing By Chunks. In Gennaro Chierchia, Pauline Jacobson, Francis J. Pelletier, Robert C. Berwick, Steven P. Abney, and Carol Tenny, editors, *Principle-Based Parsing*, volume 44, pages 257–278. Springer Netherlands, Dordrecht, 1991.

[2] Katrin Affolter, Kurt Stockinger, and Abraham Bernstein. A comparative survey of recent natural language interfaces for databases. *The VLDB Journal*, 28(5):793–819, October 2019.

[3] Arjun Akula, Rajeev Sangal, and Radhika Mamidi. A Novel Approach Towards Incorporating Context Processing Capabilities in NLIDB System. pages 1216–1222, Nagoya, Japan, October 2013.

[4] Rukshan Alexander and Prashanthi Rukshan. Natural Language Web Interface for Database (NLWIDB). In *Third International Symposium, SEUSL*, page 8, Oluvil, Sri Lanka, July 2013.

[5] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural Language Interfaces to Databases - An Introduction. *arXiv:cmp-lg/9503016*, March 1995.

[6] Ioannis Androutsopoulos. *Interfacing a Natural Language Front-End to a Relational Database*. M.Sc. Dissertation, University of Edinburgh, Department of Artificial Intelligence, 1992.

[7] Ion Androutsopoulos and Graeme Ritchie. Database Interfaces. In Robert Dale, Hermann Moisl, and Harold Somers, editors, *Handbook of Natural Language Processing*, pages 209–321. CRC Press, July 2000.

[8] Christopher Baik, H. V. Jagadish, and Yunyao Li. Bridging the Semantic Gap with SQL Query Logs in Natural Language Interfaces to Databases. *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 374–385, April 2019. arXiv: 1902.00031.

[9] Florin Brad, Radu Iacob, Ionel Hosu, and Traian Rebedea. Dataset for a Neural Natural Language Interface for Databases (NNLIDB). *arXiv:1707.03172 [cs]*, July 2017.

[10] Yohan Chandra. *Natural Language Interfaces to Databases*. MSc Dissertation, University of North Texas, Texas, US, December 2006.

[11] Jonas Chapuis. Natural Language Interfaces to Databases (NLIDB), December 2017. http://jonaschapuis.com/2017/12/natural-language-interfaces-to-databases-nlidb/.

[12] Noam Chomsky. *Syntactic Structures*. Martino, Mansfield Centre, Conn., 2015. OCLC: 934673149.

[13] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):376–386, June 1970.

[14] Ann Copestake and Karen Sparck Jones. Natural language interfaces to databases. *The Knowledge Engineering Review*, 5(4):225–249, December 1990.

[15] Facebook, Inc. GraphQL. Specification, June 2018. https://graphql.github.io/graphql-spec/June2018/.

[16] Gartner Inc. Hype Cycle Research Methodology, 2019.

[17] Alessandra Giordani. Mapping Natural Language into SQL in a NLIDB. In Epaminondas Kapetanios, Vijayan Sugumaran, and Myra Spiliopoulou, editors, *Natural Language and Information Systems*, Lecture Notes in Computer Science, pages 367–371. Springer Berlin Heidelberg, 2008.

[18] Barbara J. Grosz. TEAM: a transportable natural-language interface system. In *Proceedings of the first conference on Applied natural language processing*, pages 39–45, Santa Monica, California, 1983. Association for Computational Linguistics.

[19] Gary G Hendrix, Earl D Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum. Developing a Natural Language Interface to Complex Data. *ACM Transactions on Database Systems*, 3(2):105–147, June 1978.

[20] Dan Jurafsky and James H. Martin. *Speech and Language Processing*. draft 3rd edition, September 2018.

[21] Fei Li. *Querying RDBMS Using Natural Language*. PhD dissertation, University of Michigan, 2017.

[22] Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, September 2014.

[23] Miguel Llopis and Antonio Ferrández. How to make a natural language interface to query databases accessible to everyone: An example. *Computer Standards & Interfaces*, 35(5):470–481, September 2013.

[24] Manju Mony, Jyothi M. Rao, and Manish M. Potey. An Overview of NLIDB Approaches and Implementation for Airline Reservation System. *International Journal of Computer Applications*, 107(5):36–41, December 2014.

[25] Mrs Neelu Nihalani, Sanjay Silakari, and Mahesh Motwani. Natural language Interface for Database: A Brief review. *International Journal of Computer Science Issues*, 8, March 2011.

[26] Vesper Owei. Natural language querying of databases: an information extraction approach in the conceptual query language. *International Journal of Human-Computer Studies*, 53(4):439–492, October 2000.

[27] Jaina Patel and Jay Dave. A Survey: Natural Language Interface to Databases. *IJAERD*, 2015.

[28] Rodolfo A. Pazos R., Marco A. Aguirre L., Juan J. González B., José A. Martínez F., Joaquín Pérez O., and Andrés A. Verástegui O. Comparative study on the customization of natural language interfaces to databases. *Springerplus*, 5, April 2016.

[29] Jose F Quesada. Lekta: A Technological Framework for the Development of Conversational Enterprise Intelligent Assistants. Technical Report, Lekta.ai, Seville, October 2018.

[30] Rodolfo A. Pazos Rangel, O. Joaquín Pérez, B. Juan Javier González, Alexander Gelbukh, Grigori Sidorov, and M. Myriam J. Rodríguez. A Domain Independent Natural Language Interface to Databases Capable of Processing Complex Queries. In Alexander Gelbukh, Álvaro de Albornoz, and Hugo Terashima-Marín, editors, *MICAI 2005: Advances in Artificial Intelligence*, Lecture Notes in Computer Science, pages 833–842. Springer Berlin Heidelberg, 2005.

[31] Philip Resnik. Access to Multiple Underlying Systems in Janus:. BBN Report No. 7142, Defense Technical Information Center, Fort Belvoir, VA, September 1989.

[32] Michael G Schwern, Andy Lester, and Andy Armstrong. TAP13 - The Test Anything Protocol v13. Specification, 2007.

[33] Niculae Stratica, Leila Kosseim, and Bipin C Desai. NLIDB Templates for Semantic Parsing. page 7, 2004.

[34] Bozena H. Thompson and Frederick B. Thompson. Introducing ASK, a simple knowledgeable system. In *Proceedings of the first conference on Applied natural language processing*, Santa Monica, California, 1983. Association for Computational Linguistics.

[35] w3resource. SQL exercises on soccer Database: Exercises, Practice, Solution, 2019. https://www.w3resource.com/sql-exercises/soccer-database-exercise/index.php.

[36] David H D Warren and Fernando C. N. Pereira. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. *American Journal of Computational Linguistics*, 8(3–4):110–122, 1982.

[37] Ludwig Wittgenstein. *Philosophical investigations*. Basil Blackwell, Oxford, 1968.

[38] W. Woods, Ronald Kaplan, and Bonnie Webber. The Lunar Sciences Natural Language Information System. BBN Report No. 2378, July 1972.