Pontifícia Universidade Católica do Rio de Janeiro

Alexandre Ferreira Novello

A Novel Solution to Empower Natural Language Interfaces to Databases (NLIDB) to Handle Aggregations

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Marco Antonio Casanova

Rio de Janeiro
April 2021

Pontifícia Universidade Católica do Rio de Janeiro

Alexandre Ferreira Novello

A Novel Solution to Empower Natural Language Interfaces to Databases (NLIDB) to Handle Aggregations

Dissertation presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee.

Prof. Marco Antonio Casanova
Advisor
Departamento de Informática – PUC-Rio

Prof. Antônio Luz Furtado
Departamento de Informática – PUC-Rio

Prof. Luiz André Portes Paes Leme
UFF

Rio de Janeiro, April 23th, 2021

Alexandre Ferreira Novello

The author graduated in Computer Science from the Federal University of Rio de Janeiro (UFRJ), Rio de Janeiro, Brazil in 1999. He joined the Graduate Program in Informatics at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) in 2019.

Bibliographic data

Novello, Alexandre Ferreira

A Novel Solution to Empower Natural Language Interfaces to Databases (NLIDB) to Handle Aggregations Alexandre Ferreira Novello; advisor: Marco Antonio Casanova. – Rio de Janeiro: PUC-Rio, Departamento de Informática, 2021.

v., 85. : il. ; 29,7 cm

1. Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Interface de Linguagem Natural para Banco de Dados (Natural Language Interface to Database - NLIDB). 3. Perguntas & Respostas (Question Answering - QA). 4. Bancos de Dados. 5. Processamento de Linguagem Natural (Natural Language Processing - NLP). 6. Agregação. 7. SQL I. Casanova, Marco Antonio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my parents, Carlos Alberto Novello † and Áurea Stela Ferreira Novello †. Although they left early, they gave me my most precious asset: education. You are sorely missed.

To Lilian "Galeguinha" Lauser Albineli Novello, my friend, girlfriend, wife, companion, lover and life partner for all the understanding and support along this journey, all the others we have made, and those that are yet to come.

To Angelina "Nina Biju" Albineli Novello, my beloved daughter, for being our second sun, for all the irreverence and intelligence, and for being the person I want to serve as a model for. You make me a better person!

Acknowledgements

In the first place, I would like to thank, above all, my advisor Prof. Marco Antonio Casanova for his boundless patience, for helping me find a topic that was both challenging and suitable for a master's degree and also for having guided me in the steps towards the publication of an article.

Further, I would like to thank PUC-Rio and CAPES for providing a tuition scholarship. It was an honor to be part of the “Programa de Pós-graduação em Informática”. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

To Michael Kelly, my friend and English guru, for helping me with the language.

For my colleagues at Grupo Santa Isabel, especially Creston Fernandes, Isabel Ferraz Magalhães and Ferdinando Valle Magalhães for understanding the importance of this project for me and allowing me to dedicate the needed the time to complete it.

Thanks to all the staff, professors and my classmates from PUC-Rio, for the hours of work, study and (of course) fun. Especially to João Pedro Pinheiro, Cláudio Escudero, Leonardo Mariano Gravina Fonseca and Caio Barbosa.

To my friends Profa. Carla Amor Divino Delgado (UFRJ), Prof. Leonardo Gresta Paulino Murta (UFF) and Profa. Vanessa Braganholo Murta (UFF) for being my "academic godfathers".

To Prof. Nilton Alves Junior (CBPF) for having been my first advisor in scientific initiation and for helping me to overcome shyness and for having taught me how to speak in public.

Abstract

Novello, Alexandre Ferreira; Casanova, Marco Antonio (Advisor). **A Novel Solution to Empower Natural Language Interfaces to Databases (NLIDB) to Handle Aggregations**. Rio de Janeiro, 2021. 82p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Question Answering (QA) is a field of study dedicated to building systems that automatically answer questions asked in natural language. The translation of a question asked in natural language into a structured query (SQL or SPARQL) in a database is also known as Natural Language Interface to Database (NLIDB). NLIDB systems usually do not deal with aggregations, which can have the following elements: aggregation functions (as count, sum, average, minimum and maximum), a grouping clause (GROUP BY) and a having clause (HAVING). However, they deliver good results for normal queries. This dissertation addresses the creation of a generic module, to be used in NLIDB systems, that allows such systems to perform queries with aggregations, on the condition that the query results the NLIDB return are, or can be transformed into, a result set in the form of a table. The work covers aggregations with specificities such as ambiguities, timescale differences, aggregations in multiple attributes, the use of superlative adjectives, basic unit measure recognition, aggregations in attributes with compound names and subqueries with aggregation functions nested up to two levels.

Keywords

Natural Language Interface to Database (NLIDB); Question Answering (QA); Databases; Natural Language Processing (NLP); Aggregation; SQL.

Resumo

Novello, Alexandre Ferreira; Casanova, Marco Antonio (Orientador). **Uma Nova Solução para Capacitar Interfaces de Linguagem Natural para Bancos de Dados (NLIDB) para Lidar com Agregações**.Rio de Janeiro, 2021. 82p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Perguntas & Respostas (Question Answering - QA) é um campo de estudo dedicado à construção de sistemas que respondem automaticamente a perguntas feitas em linguagem natural. A tradução de uma pergunta feita em linguagem natural em uma consulta estruturada (SQL ou SPARQL) em um banco de dados também é conhecida como Interface de Linguagem Natural para Bancos de Dados (Natural Language Interface to Database - NLIDB). Os sistemas NLIDB geralmente não lidam com agregações, que podem ter os seguintes elementos: funções de agregação (como contagem, soma, média, mínimo e máximo), uma cláusula de agrupamento (GROUP BY) e uma cláusula HAVING. No entanto, eles fornecem bons resultados para consultas normais. Esta dissertação aborda a criação de um módulo genérico, para ser utilizado em sistemas NLIDB, que permite a tais sistemas realizar consultas com agregações, desde que os resultados da consulta que o NLIDB retorna sejam, ou possam ser transformados, em um resultado no formato tabular. O trabalho cobre agregações com especificidades como ambiguidades, diferenças de escala de tempo, agregações em atributos múltiplos, o uso de adjetivos superlativos, reconhecimento básico de unidade de medida, agregações em atributos com nomes compostos e subconsultas com funções de agregação aninhadas em até dois níveis.

Palavras-chave

Interface de Linguagem Natural para Banco de Dados (Natural Language Interface to Database - NLIDB); Perguntas & Respostas (Question Answering - QA); Bancos de Dados; Processamento de Linguagem Natural (Natural Language Processing - NLP); Agregação; SQL

Table of contents

Abbreviations

ANP	Agência Nacional de Petróleo - National Petroleum Agency
GLAMORISE	GeneraL Aggregation MOdule using a RelatIonal databaSE
JSON	JavaScript Object Notation
KwS	Keyword Search
MAS	Microsoft Academic Search
NLIDB	Natural Language Interface to Database
NLP	Natural Language Processing
NLQ	Natural Language Query
QA	Question Answering
RDBMS	Relational Database Management System
RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language

List of figures

List of tables

List of code samples

Any sufficiently advanced technology is indistinguishable from magic
Arthur C. Clarke

1.
Introduction

1.1. Natural Language Interface to Database (NLIDB)

Question Answering (QA) is a field of study dedicated to building systems that automatically answer questions asked in natural language. The translation of a question asked in natural language into a structured query in a database is also known as Natural Language Interface to Database (NLIDB).

The general functioning of any NLIDB is essentially that demonstrated in the architecture of the Figure 1. Initially, the user types a query *N* in natural language in the user interface. The NLIDB transforms *N* into a structured query *Q*, being SQL in the case of a RDBMS or SPARQL in the case of a triplestore, and a result set *R* is returned to the NLIDB and then presented in the user interface.

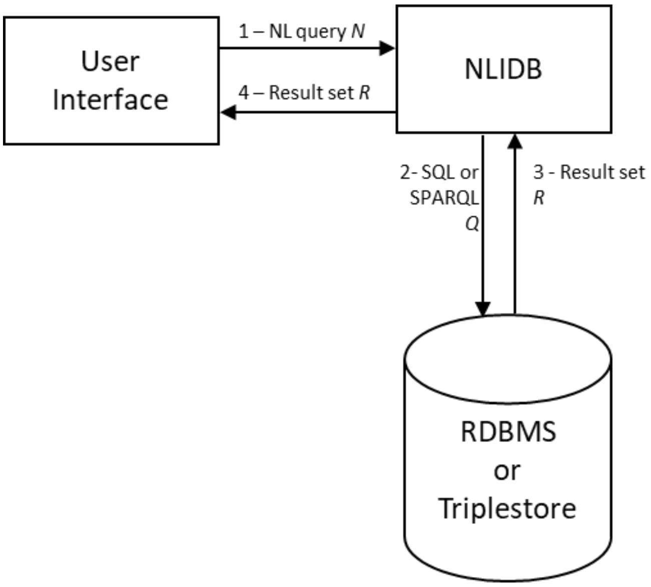


Figure 1 – General NLIDB Architecture

To obtain the structured query (SQL or SPARQL) used to query the database that will give the answer, the question in natural language goes through several processing steps such as: (1) question analysis; (2) phrase mapping; (3) disambiguation and (4) query construction.

1.2. Aggregation

Let us assume that the following database table is used in our examples. The knowledge domain of this table is the production of oil fields in Brazil. The data is released by the National Petroleum Agency (ANP)[1].

[1] http://www.anp.gov.br/

Name	Type
FIELD	TEXT
BASIN	TEXT
STATE	TEXT
OPERATOR	TEXT
CONTRACT_NUMBER	TEXT
OIL_PRODUCTION	REAL
GAS_PRODUCTION	REAL
MONTH	INTEGER
YEAR	INTEGER

Table 1 – ANP Table

Aggregation queries could have the following elements: aggregation functions (as count, sum, average, minimum and maximum), a grouping clause (GROUP BY) and a having clause (HAVING). For example, the natural language question: “How many fields are there in Paraná? ” would be translated to the following SQL query:

```
SELECT COUNT(DISTINCT field)
FROM anp
WHERE lower(state) like '%paraná%'
```

Listing 1 – SQL with just aggregation function

In this example, the aggregation function is count. Another example involving grouping could be: “What was the maximum production of oil in the state of Ceará per field?” The SQL query would be:

```
SELECT field, MAX(oil_production) AS max_oil_production
FROM anp
WHERE lower(state) like '%ceará%'
GROUP BY field
ORDER BY field
```

Listing 2 – SQL with aggregation function and grouping

Note that this query has the additional clause GROUP BY, that is, in addition to the aggregation function, it also uses grouping. The ORDER BY is just a small addition to facilitate visualization of the results.

Another example involving the clause HAVING could be: “What was the mean gas production per field with production greater than 100 cubic meters?” In this case the SQL query would be:

```
SELECT field, AVG(gas_production) AS avg_gas_production
FROM anp
GROUP BY field
HAVING AVG(gas_production) > 100
ORDER BY field
```

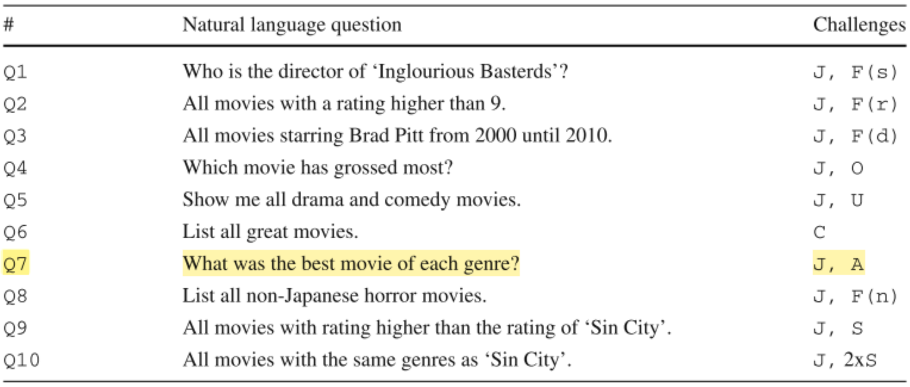
Listing 3 – SQL with aggregation function, grouping and having clause

The attentive reader will notice that the way the data is stored in the database also influences the answer given to the natural language questions. In this way, the modeling of the database must be designed in order to meet the responses expected by users and, at the same time, the lay user must have some notion of what granularity and how the information is stored in the database.

1.3. Motivation

One of the less well-solved tasks in NLIDBs is the treatment of questions with aggregation, especially when the question is on another timescale in relation to the stored data and it is necessary to perform a conversion. For example, consider the question: "What was the average yearly compensation for employees in 2020?" If the stored data related to compensation is on a weekly scale, it is necessary to understand that the question is on an annual scale and perform the equivalent operation. Note that, in this case, it is not enough to multiply the average weekly remuneration by 52, as the salary may have changed over the year as well as other sporadic events, such as vacations, or events with specific periodicity, such as bonuses. It is necessary to filter the sum of all 2020 tuples of compensation per employee and only then perform the average.

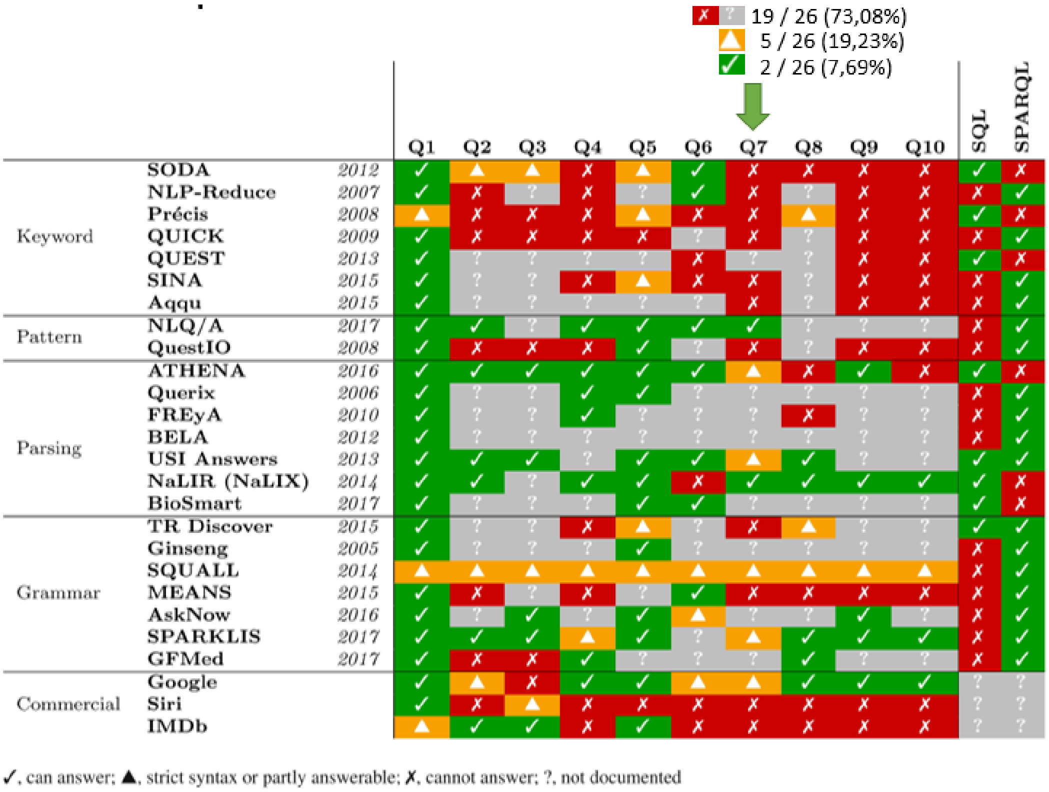
How can we substantiate the statement that one of the less well-solved tasks in NLIDBs is the treatment of questions with aggregation? To support this statement, we present a survey that compares 26 different NLIDBs [Affolter et al. 2019]. To validate this comparison, 10 questions in natural language were asked that cover different possible aspects of a structured query (join, filter, aggregation, ordering, union, subquery and concept) shown below. Note that only (Q7) is a question with aggregation.



#	Natural language question	Challenges
Q1	Who is the director of ‘Inglourious Basterds’?	J, F(s)
Q2	All movies with a rating higher than 9.	J, F(r)
Q3	All movies starring Brad Pitt from 2000 until 2010.	J, F(d)
Q4	Which movie has grossed most?	J, O
Q5	Show me all drama and comedy movies.	J, U
Q6	List all great movies.	C
Q7	What was the best movie of each genre?	J, A
Q8	List all non-Japanese horror movies.	J, F(n)
Q9	All movies with rating higher than the rating of ‘Sin City’.	J, S
Q10	All movies with the same genres as ‘Sin City’.	J, 2xS

Figure 2 – Benchmark questions [Affolter et al. 2019]

With these questions in mind, a comparative analysis of the NLIDBs was made, demonstrating which were capable of answering each question correctly, or partially or with a reduced syntax, and which could not. In some cases, it was unclear in the paper if they could answer or not. The result is shown in Figure 3. If we look at the column referring to Q7, we will see that, of the 26 NLIDBs, only 2 (7.69%) managed to answer the aggregation question correctly, which supports the claim that most NLIDBs do not handle aggregation questions well.



🟥✗ ▨? 19 / 26 (73,08%)
▲ 5 / 26 (19,23%)
✓ 2 / 26 (7,69%)

			Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	SQL	SPARQL
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	✓	✓	✓	?	✓	✓	▲	✓	?	?	✓	✓
	NaLIR (NaLIX)	2014	✓	✓	?	✓	✓	✗	✓	✓	✓	✓	✓	✗
	BioSmart	2017	✓	?	?	?	✓	✓	?	?	?	?	✓	✗
Grammar	TR Discover	2015	✓	?	?	✗	▲	?	✗	▲	?	?	✓	✓
	Ginseng	2005	✓	?	?	?	✓	?	?	?	?	?	✗	✓
	SQUALL	2014	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	✗	✓
	MEANS	2015	✓	✗	?	✗	?	✓	✗	✗	✗	✗	✗	✓
	AskNow	2016	✓	?	✓	?	✓	▲	?	?	✓	?	✗	✓
	SPARKLIS	2017	✓	✓	✓	▲	✓	?	▲	✓	✓	✓	✗	✓
	GFMed	2017	✓	✗	✗	✓	?	?	?	✓	?	?	✗	✓
Commercial	Google		✓	▲	✗	✓	✓	▲	▲	✓	✓	✓	?	?
	Siri		✓	✗	▲	✗	✗	✗	✗	✗	✗	✗	?	?
	IMDb		▲	✓	✓	✗	✓	✗	✗	✗	✗	✗	?	?

✓, can answer; ▲, strict syntax or partly answerable; ✗, cannot answer; ?, not documented

Figure 3 – NLIDBs comparison & benchmark questions [Affolter et al. 2019]

Furthermore, the survey compared the 10 benchmark questions with corpora questions: Yahoo! QA Corpus L62 (more than 4 million questions) and GeoData250 (250 questions against a database). It found that found that the majority of the corpora questions belonged to the Q1 and the Q4 types. Note that Figure 3 shows that the majority of the NLIDBs were able to answer Q1 correctly and one third were able to answer Q4 correctly. What if it were possible to provide NLIDBs already capable of answering most questions with the added ability to answer questions with aggregation? This is the proposition of GLAMORISE (GeneraL Aggregation MOdule using a RelatIonal databaSE).

1.4. Contributions

NLIDB systems usually do not deal with aggregations, but they produce good results for normal queries. The contribution of this dissertation is the creation of a generic module, called GLAMORISE, to be used in NLIDB systems. This module allows NLIDB systems to perform queries with aggregations, on the condition that the result of the NLIDB is, or can be transformed into, a result set in the form of a table. Hence, it can also be used with triplestore (RDF store) NLIDBs with the proviso that the result is presented in a tabular format. The tabular format is returned to GLAMORISE in JSON format. The dissertation addresses some aggregations with specificities, including ambiguities, timescale differences, aggregations in multiple attributes, the use of superlative adjectives, basic unit measure recognition, aggregations in attributes with compound names and subqueries with aggregation functions nested up to two levels.

To test the proposed approach, GLAMORISE was first integrated with a mock NLIDB, which will be described in Section 4.1, and tested with 22 static questions as a proof-of-concept. Then, GLAMORISE was integrated with NaLIR [Li and Jagadish, H. V. 2014], a real NLIDB. As shown in Figure 3, NaLIR is the NLIDB that performed best in this comparison, including answering questions with aggregation. The aggregation layer was removed from NaLIR, leaving GLAMORISE to address this issue, while NaLIR addressed the remaining issues involved in an NLIDB. This integration is described in Section 4.2.1.

To validate the results of the GLAMORISE/NaLIR integration, we used two datasets: first, we used the 22 questions developed for the ANP dataset as a proof-of-concept; then, we adopted 17 questions from the Microsoft Academic Search (MAS). The original NaLIR work considered 194 NLQs[2], but one of them was duplicated (‘return me the author who has the highest number of papers containing keyword "Relational Database".’). Of these, 99 NLQs referred to questions with aggregation but, within these, the linguistic and structural patterns recurred repeatedly. So, we chose 17 NLQs that represented the universe of questions contained in the article.

Another experiment that is being carried out is the integration of GLAMORISE with another NLIDB called DANKE [García 2020; Izquierdo et al. 2018, 2020; Torres Izquierdo et al. 2020], which is a NLIDB that lacks the ability to answer aggregation questions. DANKE is an NLIDB of the Keyword Search (KwS) type and, as such, are unable to answer questions with aggregation [Affolter et al. 2019]. This is described in the Section 4.2.2.

1.5. Dissertation structure

The rest of this dissertation is organized as follows. Chapter 2 is a literature review. Chapter 3 describes our solution, its features and limitations. Chapter 4 evaluates performance against benchmark questions. Chapter 5 contains our conclusions. Finally, Chapter 0 contains the references.

[2] https://raw.githubusercontent.com/umich-dbgroup/NaLIR/master/mas_all.nlqs

2.
Literature Review

SQAK (SQL Aggregates using Keywords) [Tata and Lohman 2008] is a framework that allows users to perform queries with aggregations using only keywords, with no knowledge of the database schema or SQL. The concept of Simple Query Network (SQN), which is similar to the Steiner Tree, but with better results for this purpose, was created. A greed algorithm was developed to find the minimal SQN, since this is an NP-Complete problem, and used to build the SQL. Our work and SQAK deals with similar problems, aiming at the use of keywords for the final translation into SQL. The difference is that their work as well as others that we will refer to in this section are complete and monolithic NLIDBs, while ours aims at enabling existing NLIDBs to handle aggregations, which they do not perform well.

NaLIR [Li and Jagadish, H. V. 2014; Li and Jagadish 2016; Li and Jagadish, H. V 2014] is a generic NLIDB capable of handling aggregations, nesting and various types of joins. The Stanford NLP Parser is used to convert from natural language into a parser tree. The approach followed is to get feedback from the user and return the adjusted parse trees back to the user in the form of natural language so that they can select the natural language question that makes the most sense or revise accordingly. In our view, this exchange of information jeopardizes the user experience, as they have the impression that are carrying out work that should be done by the NLIDB, regardless of the extent to which it ensures that the resulting query is correct after the adjustments. However, the version we use of the NaLIR integrated in our work does not have the layer called interactive communicator, returning an answer automatically to the user, since in our work, we prefer the approach of generating a structured query automatically from the natural language query. Still, NaLIR has good results and is considered one of the best academic NLIDBs.

In [Gupta et al. 2012] a novel approach was presented to building NLIDB based on dependency trees with the use of Computational Paninian Grammar (CPG) [Bharati et al. 2014] in which the relationships are syntactic-semantic. CPG was originally developed for Indian languages and afterwards gained an English version. They argued that the use of this technique makes the trees more semantic than other kinds of dependency trees, thus making them easier to map to a SQL. In the following article [Gupta and Sangal 2013], the framework was extended to handle aggregation processing with different types of aggregation operations in natural language, including quantitative and qualitative aggregations, and those combining quantifiers or relational operators with aggregations. A separate layer in the querying process was devised: first the SQL query is generated and processed in the RDBMS without the aggregation and then the aggregation is processed in the returned result set. The whole concept is explained in detail in Gupta’s master’s dissertation [Abhijeet Gupta 2013]. This work served as an inspiration regarding the isolation and classification of the parts that compose the aggregation and the processing of the aggregation in the returned result set described in Section 3.

Another work was also developed that followed the two-stage strategy. First, an NLIDB was created without the ability to process aggregations (or subqueries) [Pazos R et al. 2016], called ITCM NLIDB, and a second work [Pazos R et al. 2018] added a module capable of carrying out these activities. The NLIDB kernel is composed of three main modules: a lexical analyzer (tags the words in the lexicon with their syntactic categories); a syntactic module (leaves only one syntactic category for each lexical component and disregards irrelevant ones); and a semantic module (maps the result of the previous steps in tables and columns in the database). Although simpler than previous works, the main contribution of this work was to identify recurring problems in how aggregations (and subqueries) are stated in natural language, and to propose solutions to these problems. The problem is that, as in NaLIR, they did not seek an automatic solution, but returned the ambiguities for the user to resolve, which, in our opinion, makes the process less user-friendly. Nevertheless, this work was useful as it confirmed various recurring problems when dealing with queries in natural language with aggregations, which were also identified by us.

One survey on the subject also entered our literature search. [Affolter et al. 2019] was published in the 2019 VLDB. It has already been commented on in Section 1.3 when discussing the motivation for our work. In addition to the comparison of the 26 different NLIDBs, the contribution of this paper to our work was the confirmation that questions with aggregation are in fact a problem that NLIDBs have difficulty dealing with.

While ours and most of the works related to NLIDB that deal with the problem of aggregation focus on questions, an interesting paper is [Pinheiro et al. 2020]. Rather than focusing on questions with aggregation, aggregation is leveraged as one of the techniques to be employed when presenting answers to users should there be a sizable result set. Moreover, another technique used in the presence of a large result set is to establish a dialogue with the user in order to reduce the size of said set by refining it.

Two additional papers that are tangential to our undertaking were also read. These handle natural language queries converted into structured queries to address other problems. TiQi, solution for Software Traceability, enables users to build natural language trace queries that are converted into SQL [Pruski et al. 2015]. In SpeakQL the approach is to facilitate the construction of SQL queries using spoken queries in mobile devices [Shah et al. 2019].

3. GLAMORISE – A Proposed Solution to Process Aggregations

3.1. Black Box Integration Architecture

As mentioned previously, NLIDB systems usually do not deal with aggregations, but they return good results for normal queries. In this work, we propose a generic module, called GLAMORISE, to be used in NLIDB systems. This module allows NLIDB systems to perform queries with aggregations, as long as the result is, or can be transformed into, a result set in the form of a table.

The integration of GLAMORISE with a NLIDB can be done in two ways. If it is possible to have access to the NLIDB source code, the proposed GLAMORISE functionalities can be integrated into its source code. This solution is called *white box* and the architecture would be the same as in Figure 1. If it is not possible to have access to the source code of the integrated NLIDB, a *black box* solution should be adopted, as shown in Figure 4.

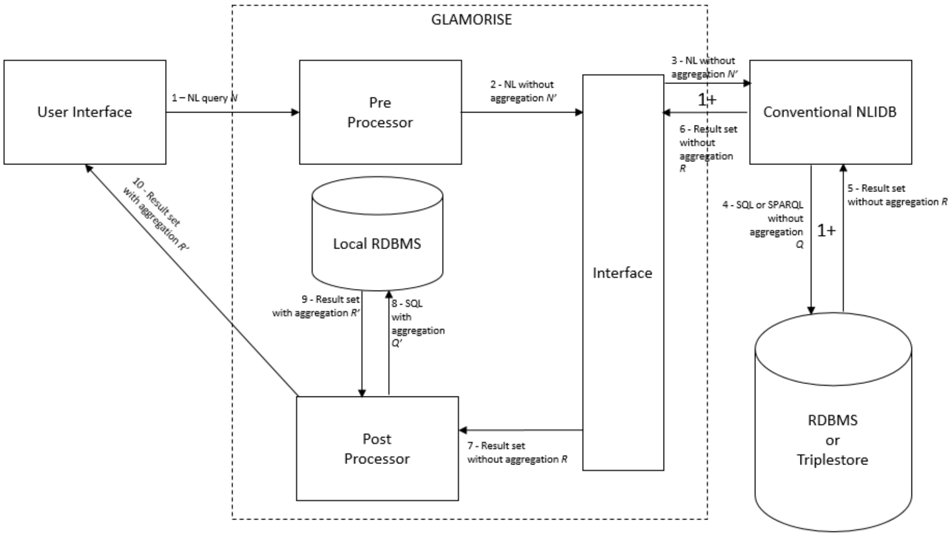

GLAMORISE
User Interface
1 – NL query N
Pre Processor
2 - NL without aggregation N'
Interface
3 - NL without aggregation N'
1+
6 - Result set without aggregation R
Conventional NLIDB
4 - SQL or SPARQL without aggregation Q
1+
5 - Result set without aggregation R
RDBMS or Triplestore
Local RDBMS
9 - Result set with aggregation R'
8 - SQL with aggregation Q'
Post Processor
7 - Result set without aggregation R
10 - Result set with aggregation R'

Figure 4 – GLAMORISE black box integration architecture.

Initially, the user types a query *N* in natural language in the user interface (arrow 1). The GLAMORISE **Preprocessor** removes the aggregation elements and transforms *N* into a query *N'* without aggregation, in natural language, and registers all the elements related to the aggregation, to be subsequently used by the **Postprocessor**. Then, *N'* is sent to the **Interface** layer (arrow 2), which is responsible for the integration with the conventional NLIDB (arrow 3). After which, the query is processed by the conventional NLIDB and converted into a structured query *Q* (arrow 4), being SQL in the case of a RDBMS or SPARQL in the case of a triplestore, and a result set *R* without any aggregations is returned in a tabular format (arrows 5, 6 and 7). Additionally, the metadata of the data result set can be retrieved to process more intricate questions. To improve the result given by conventional NLIDB and depending on the implementation of the integration made in the **Interface** layer, the steps corresponding to arrows 3, 4, 5 and 6 could be iterated more than once until the result achieved is satisfactory. Following this, the GLAMORISE **Postprocessor** stores *R* as a table in a local SQLite RDBMS, processes the aggregation over the stored result set *R* by creating a SQL query *Q'*(arrow 8), resulting in the final result set with aggregations *R'*(arrow 9), and presents *R'* to the user interface (arrow 10).

3.1.1. Preprocessor

The purpose of the **Preprocessor** is to map the keywords in natural language to the respective aggregation functions, identify whether the query in natural language also has a grouping clause (GROUP BY) or a having clause (HAVING).

These keywords are removed or substituted from the query to guarantee that the conventional NLIDB will not be confused by their presence, leading to incorrect mapping. More examples will be shown later on in this section.

3.1.2. Interface

The Interface layer is responsible for the integration with the conventional NLIDB. Its implementation is dependent on the underlying conventional NLIDB. More details of the implementation will be described in Section 3.4.

3.1.3. Postprocessor

The **Postprocessor** is responsible for constructing the SQL query, for analyzing the metadata saved in the **Preprocessor** stage and including the aggregation functions (sum, max, min, avg and count), as well as recognizing the fields in which these functions should be applied in the received result set. Then, an identical process is undertaken for grouping clause (GROUP BY), reading the metadata to determine if there is a grouping, which fields are involved, and mapping them in the result set.

All fields identified as belonging to the GROUP BY clause are inserted in the SELECT and the ORDER BY clauses, the latter for the user's convenience only.

In this layer we also detect if there is a having clause (HAVING) and its conditions.

Another step is conducted to analyze any timescales and nested aggregation that should be converted to a subquery.

It is important to say that it is possible that some fields are only found by the conventional NLIDB. For these fields, this layer also analyzes their inclusion in the SELECT clause of and GROUP BY, the latter if it exists.

3.2. Aggregation Types

We first note that the version of GLAMORISE considered in this dissertation is prepared to accept questions only in English. Nothing prevents it from being extended to other languages, since the linguistic patterns used exist in most languages and there are packages of different languages in the NLP libraries used by GLAMORISE and integrated NLIDBs.

This section describes all specificities related to the aggregations covered by GLAMORISE. All examples are based on the sample table shown in Table 1 of Section 1.2.

Examples of how to configure these patterns in the GLAMORISE configuration file are shown in Appendix I.

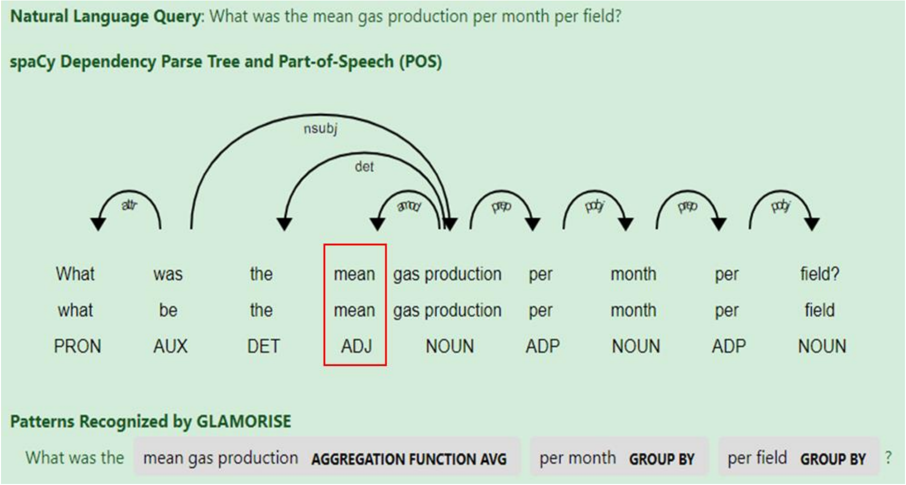
3.2.1. Aggregation Ambiguities

We must separate the two types of ambiguities that may occur. First, there are ambiguities that can be resolved directly by the NLIDB, such as a keyword that is mapped to more than one element of the database (table, attribute, data). Second, there are those which involve aggregation, i.e., a word that should be understood as part of the aggregation, but that can be of another syntactic-semantic nature. Any ambiguity to be addressed by the NLIDB will be resolved without our module even being aware of it. For ambiguities directly related to aggregations, the solution applied is to recognize the true nature of the word according to the pattern of the utterance.

The two sentences below have the same intent and should be translated to the same query. The only difference between them is the word order:

What was the mean gas production per month per field?
What was the per month mean gas production per field?

We adopt spaCy[3] to parse the NL sentences. spaCy outperforms other libraries, such as NLTK and the Stanford NLP Parser, but it is not perfect. The following two figures show the spaCy parse trees of these sentences and the interpretation of our system:



Natural Language Query: What was the mean gas production per month per field?

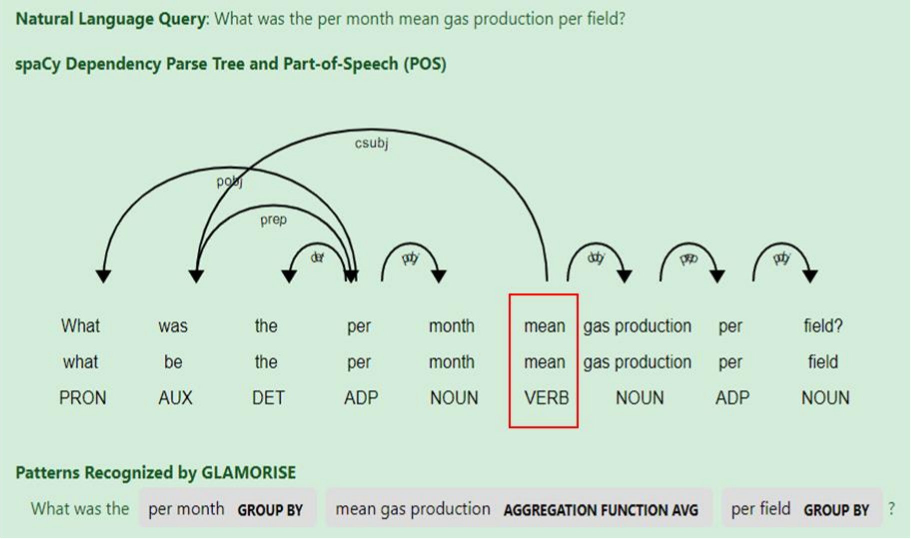
spaCy Dependency Parse Tree and Part-of-Speech (POS)

What	was	the	mean	gas production	per	month	per	field?
what	be	the	mean	gas production	per	month	per	field
PRON	AUX	DET	ADJ	NOUN	ADP	NOUN	ADP	NOUN

Patterns Recognized by GLAMORISE

What was the | mean gas production **AGGREGATION FUNCTION AVG** | per month **GROUP BY** | per field **GROUP BY** | ?

Figure 5 – Aggregation ambiguity – mean as adjective.



Natural Language Query: What was the per month mean gas production per field?

spaCy Dependency Parse Tree and Part-of-Speech (POS)

What	was	the	per	month	mean	gas production	per	field?
what	be	the	per	month	mean	gas production	per	field
PRON	AUX	DET	ADP	NOUN	VERB	NOUN	ADP	NOUN

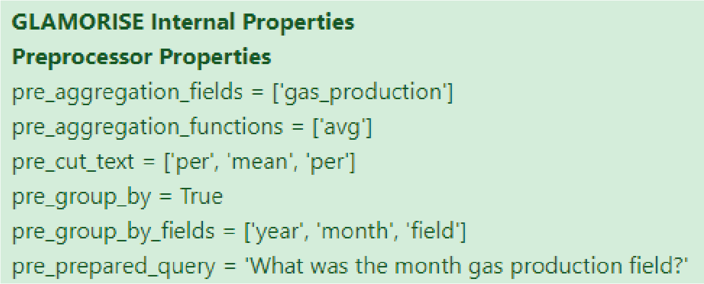
Patterns Recognized by GLAMORISE

What was the | per month **GROUP BY** | mean gas production **AGGREGATION FUNCTION AVG** | per field **GROUP BY** | ?

[3] https://spacy.io/

Figure 6 – Aggregation ambiguity – mean as verb.

The word “mean” was recognized in the first parse tree as ADJ, which means an adjective; but it was wrongly recognized in the second parse tree (corresponding to the second question) as VERB, which means a verb, despite the **Preprocessor** recognizing the true intent of “mean” and converting it to an avg (average) aggregation function, as shown in the figure below.



GLAMORISE Internal Properties
Preprocessor Properties

```
pre_aggregation_fields = ['gas_production']
pre_aggregation_functions = ['avg']
pre_cut_text = ['per', 'mean', 'per']
pre_group_by = True
pre_group_by_fields = ['year', 'month', 'field']
pre_prepared_query = 'What was the month gas production field?'
```

Figure 7 – Aggregation ambiguity – Preprocessor recognizes correctly.

This behavior is possible because, in our experiments, we realized that the best results were not in the analysis of the Part-Of-Speech (POS) of the keyword, in the above case "mean", but the POS of the words that come after the keyword. The mechanism of operation and configuration of keywords and patterns will be explained in detail in Section 3.6.2.

We will use this case to demonstrate how GLAMORISE settings in JSON work. To recognize "mean" we have a rule with the following linguistic pattern.

```
"mean example": {
  "reserved_words": ["mean"],
  "pre_aggregation_functions": "avg",
  "pre_cut_text": true
}
```

Listing 4 – “mean” rule in GLAMORISE JSON configuration file

The first declaration is the name of the rule, in this case "mean example". Then, we have the definition of the reserved words of this rule, using the "reserved_words" parameter; in the example, only "mean" is set as a reserved word. Next, the "pre_aggregation_functions" parameter indicates which aggregation functions to use, in the case "avg". Finally, since the "pre_cut_text" parameter is set to true, GLAMORISE cuts the reserved word "mean" so as not to confuse the integrated NLIDB that is not prepared to deal with aggregations.

Within each rule, after each reserved word, a set of words is expected, and they must have certain Part-Of-Speech (POS) and relations in the dependency tree. This is indicated through the "specific_pattern" parameter, or by the parameter "default_pattern", which is valid for all rules that do not have a "specific_pattern" defined, as is the case with the rule in Listing 5. The "default_pattern" is defined below.

```
"default_pattern": [{"POS": "ADV", "OP": "*"},
                    {"POS": "ADJ", "OP": "*"},
                    {"POS": "NOUN", "LOWER": {"NOT_IN":
       ["number"]}}
                   ]
```

Listing 6 – default pattern explained

This parameter defines the standard way in which a field is found. Used in conjunction with the rule of Listing 7, this use of the default pattern says that the reserved word (or keyword) “mean” must be followed by an optional adverb ({"POS": "ADV", "OP": "*"}), an optional adjective ({"POS": "ADJ", "OP": "*"}) or a noun ({"POS": "NOUN", "LOWER": {"NOT_IN": ["number"]}}), which is the field to be identified. This noun can be simple or compound (as discussed later). The declaration ("LOWER": {"NOT_IN": ["number"]}) indicates that "number" and "number of" are reserved words that are often used to mean a counting function and the word "number" is a noun, so it should not be considered for field match purposes. For example, in the part of the sentence "mean gas production", the "mean" is taken from the "reserved_words" of the rule in Listing 8. This rule uses "default_pattern" because it does not have "specific_pattern" defined and "gas production" matches the “default_pattern”, because it is a noun. Remembering that "default_pattern" waits for an adverb (optional), an adjective (optional) and a noun, which "gas production" matches. Therefore, the rule as a whole works, as "mean" matches "reserved_words" and "gas production" matches the “default_pattern”.

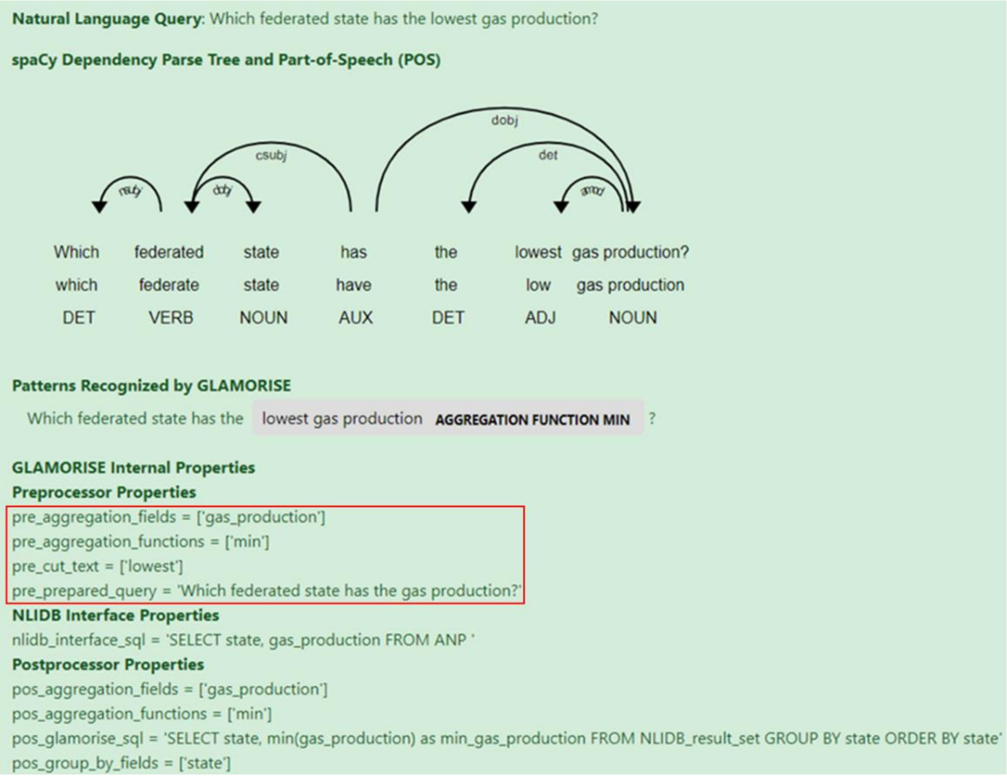
Examples of how to configure these patterns in the GLAMORISE configuration file are shown in Appendix I and the parameters of the rules are described in the Section 3.6.2.

The second kind of ambiguity that we deal with uses terms such as “greater than”, “more than”, “less than”. This kind of expression could be translated into two different clauses in SQL depending on the database schema. If the condition is directly related to the value of a field in a table, the condition is translated into a WHERE clause. If it is related to any grouping action, the condition is translated into a HAVING clause. As we do not have access to the database schema that NLIDB will query, we do not know in advance which of the two cases we are dealing with. Bearing that in mind, we save the expression to be used later by the **Postprocessor** in the HAVING clause. We also do not remove the expression from the query communicated to the NLIDB since, if the

NLIDB can interpret it, then the expression has an impact on the WHERE clause, otherwise the NLIDB will ignore the expression.

3.2.2. Superlative Adjectives

Superlative adjectives could be suppressed and, depending on the type of superlative, a *min* or *max* function is added to the metadata of the aggregation functions; the respective aggregate field is also added. The superlative adjective is then removed from the query to not confuse the NLIDB with a term that it cannot handle. An example is presented below:



Natural Language Query: Which federated state has the lowest gas production?

spaCy Dependency Parse Tree and Part-of-Speech (POS)

dobj
csubj
det
nsubj
dobj
amod
Which federated state has the lowest gas production?
which federate state have the low gas production
DET VERB NOUN AUX DET ADJ NOUN

Patterns Recognized by GLAMORISE

Which federated state has the lowest gas production **AGGREGATION FUNCTION MIN** ?

GLAMORISE Internal Properties

Preprocessor Properties

```
pre_aggregation_fields = ['gas_production']
pre_aggregation_functions = ['min']
pre_cut_text = ['lowest']
pre_prepared_query = 'Which federated state has the gas production?'
```

NLIDB Interface Properties

```
nlidb_interface_sql = 'SELECT state, gas_production FROM ANP '
```

Postprocessor Properties

```
pos_aggregation_fields = ['gas_production']
pos_aggregation_functions = ['min']
pos_glamorise_sql = 'SELECT state, min(gas_production) as min_gas_production FROM NLIDB_result_set GROUP BY state ORDER BY state'
pos_group_by_fields = ['state']
```

Figure 8 – Superlative adjective example

The *min* aggregation function is identified due to the presence of the superlative adjective “lowest”; also, “gas production” is identified as the related aggregate field due to the pattern configuration described in Section 3.6.2.

3.2.3. Multiple Attribute Aggregations

An aggregation may have more than one attribute. For example, consider again the questions *“What was the mean gas production per month per field?”* and *“What was the per month mean gas production per field?”*. The **Postprocessor** will construct the following SQL query:

```
SELECT year, month, field, AVG(gas_production) AS
avg_gas_production
FROM nlidb_result_set
GROUP BY year, month, field
ORDER BY year, month, field
```

Listing 9 – SQL with multiple attribute aggregations

Another variation for the question would be: *“What was the mean gas production per month and field?”*. Usually, the keyword “and” is used for conditions in the WHERE clause, but the **Preprocessor** could be configured to understand that an “and” preceded by a “per | by *field* is a keyword for the GROUP BY clause.

3.2.4. Aggregations of attributes with compound names

Attributes may be expressed as compound names such as “oil production”. A conventional NLIDB must deal with compound names to build the query without aggregations. But, this kind of attribute name may also be present in the aggregation functions, grouping or having clauses. To handle this, the **Preprocessor** resorts to the use of a spaCy parse tree to identify nouns that are compound names. It also identifies, through patterns, the use of "of" connecting two nouns. Thereafter, the **Interface** attempts to associate the keywords with the NLIDB result set to identify which keywords match the corresponding attributes within the NLIDB result set.

In fact, this treatment is done before the steps realized by GLAMORISE described in Figure 4. spaCy natively has a pipeline for treating NLQs. It also allows you to create your own processing steps and include them in this pipeline. To address this issue and simplify the treatment of compound nouns made by the later stages of GLAMORISE, we created a processing step and added it to the end of the spaCy pipeline. This step converts compound nouns into a single noun. We can see below how the sentence looks in the standard spaCy tree in Figure X and how it looks after the modification made by GLAMORISE in Figure Y, This technique simplifies the configuration of GLAMORISE, described in Section 3.6.2.

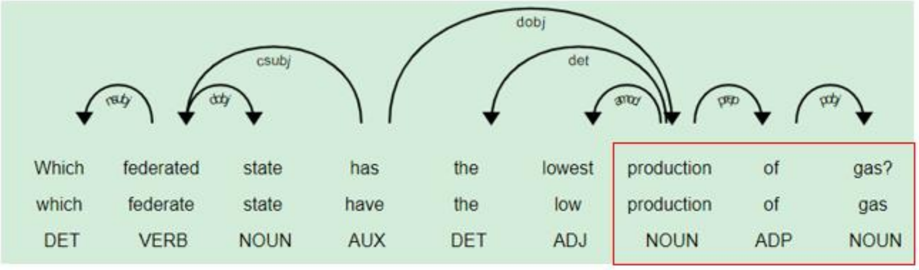


Figure 9 – Standard Spacy Pipeline

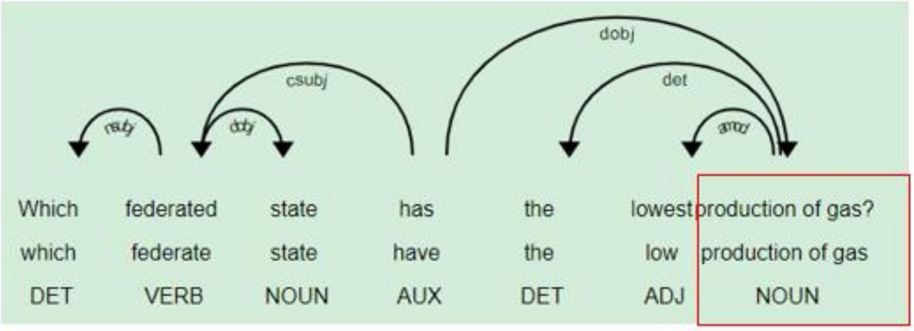


Figure 10 – Spacy Pipeline Customized by GLAMORISE

3.2.5. Simple Recognition of Units of Measurement

Using the same technique, another step is added to the end of spaCy pipeline to deal with the simple recognition of units of measurement.

The presence of units of measurement can confuse or make the GLAMORISE configuration very laborious, since the format that they take varies greatly. Hence, GLAMORISE allows units of measurement to be included in the configuration file to be recognized later. It is beyond the scope of this work, although it is an interesting topic, to perform any manipulation of units of measurement, such as converting from one unit to another. For example, the question could be asked in one unit of measurement and the data could be stored in in another unit of measurement, which would require a conversion to deliver the correct answer.

3.2.6. Subquery - Aggregations with timescale differences

The only advanced treatment of units of measurement considered in this dissertation is when we have time scales with each unit of time having its own column in the database, for example, one column for year, another for month, another for day, etc.

To the best of our knowledge, this is a special condition in aggregation for which we did not find a solution in other works. The problem becomes apparent when the data is stored in one timescale and the question is asked in another. Going back to our ANP table, an example could be: *"What was the average yearly production of oil in the state of Alagoas?"* The problem would arise if the data stored in the table is on a monthly basis. The ANP table, shown in Table 1, has two attributes, one for the year and another for the month, in addition to production (oil or gas). That is, each tuple associates an oil production value to one year and one month. The equivalent SQL query on Oracle that accepts nested aggregate functions would be:

```
SELECT AVG(SUM(oil_production))
    as avg_sum_oil_production
FROM nlidb_result_set
WHERE state = 'Alagoas'
GROUP BY year
```

Listing 10 – SQL with nested aggregation functions (Oracle standard)

The reader will notice that this query is different from the previous examples. Namely, there are two aggregation functions: the first one performs the sum of the oil production grouped by the attribute year, while the second computes the average over all years.

For convenience, the first implementation of GLAMORISE (see Section 3.3) uses SQLite[4] to store the metadata and process the aggregation in the result set. Since, at the moment, SQLite does not support nested aggregation functions, such as “AVG(SUM(*field*))”, the above NL query has to be translated to:

```
SELECT AVG(sum_oil_production)
    as avg_sum_oil_production
FROM(SELECT SUM(oil_production)
    as sum_oil_production
FROM nlidb_result_set
WHERE state = 'Alagoas'
GROUP BY year)
```

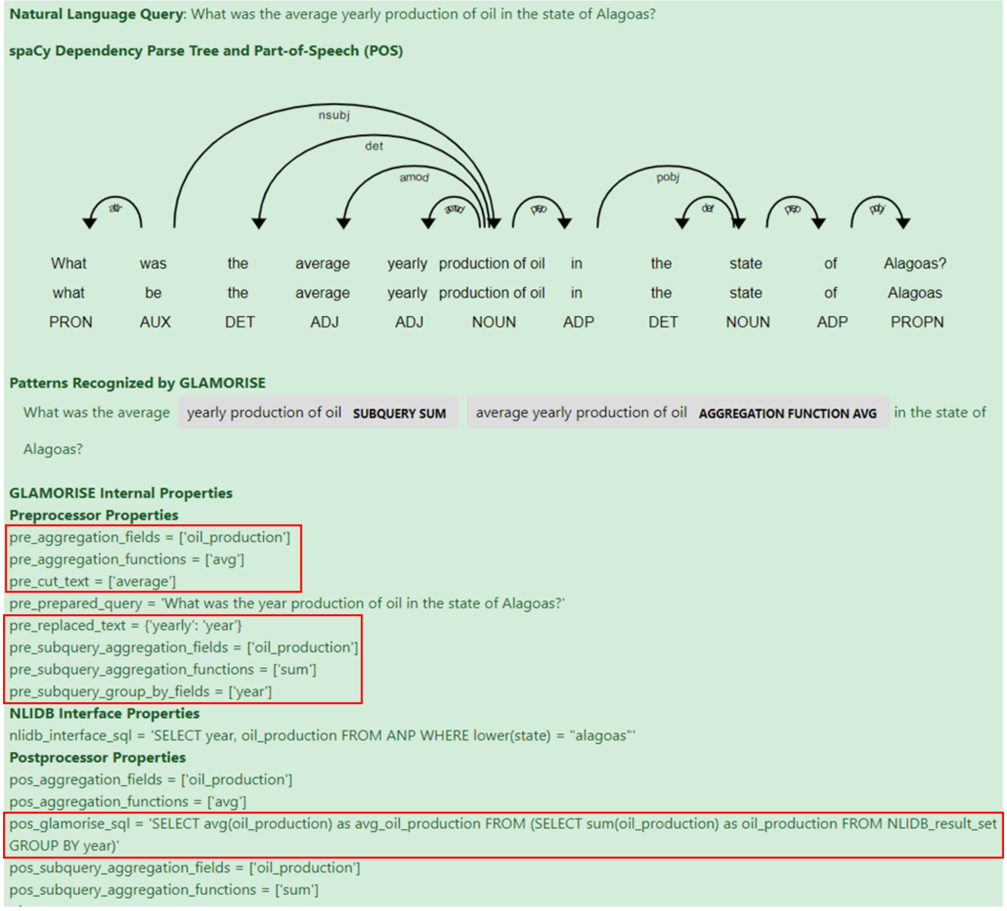
Listing 11 – SQL with subquery instead of nested aggregation functions

The **Preprocessor** converts the adjective, in the case of the example, "yearly", to its corresponding noun, in this case, "year". When it receives the NLIDB result set for this type of question, it also receives the result set with information regarding the timescale in which the data is stored (daily, monthly, yearly, etc.) depending on the columns returned. If the question were asked in a different scale, the **Postprocessor** would translate the aggregation accordingly (SUM(*field*) and GROUP BY).

Erro! Fonte de referência não encontrada. Figure 11 shows how the **Preprocessor** recognizes the sentences and separates the "average" interpretation, which is the normal aggregation that will be

[4] https://www.sqlite.org/

made, from the "yearly" interpretation, which is the timescale aggregation that will be made, under the form of a subquery, depending on the timescale that is stored in the database.



Natural Language Query: What was the average yearly production of oil in the state of Alagoas?

spaCy Dependency Parse Tree and Part-of-Speech (POS)

Patterns Recognized by GLAMORISE

What was the average yearly production of oil **SUBQUERY SUM** average yearly production of oil **AGGREGATION FUNCTION AVG** in the state of Alagoas?

GLAMORISE Internal Properties

Preprocessor Properties

pre_aggregation_fields = ['oil_production']
pre_aggregation_functions = ['avg']
pre_cut_text = ['average']
pre_prepared_query = 'What was the year production of oil in the state of Alagoas?'
pre_replaced_text = {'yearly': 'year'}
pre_subquery_aggregation_fields = ['oil_production']
pre_subquery_aggregation_functions = ['sum']
pre_subquery_group_by_fields = ['year']

NLIDB Interface Properties

nlidb_interface_sql = 'SELECT year, oil_production FROM ANP WHERE lower(state) = "alagoas"'

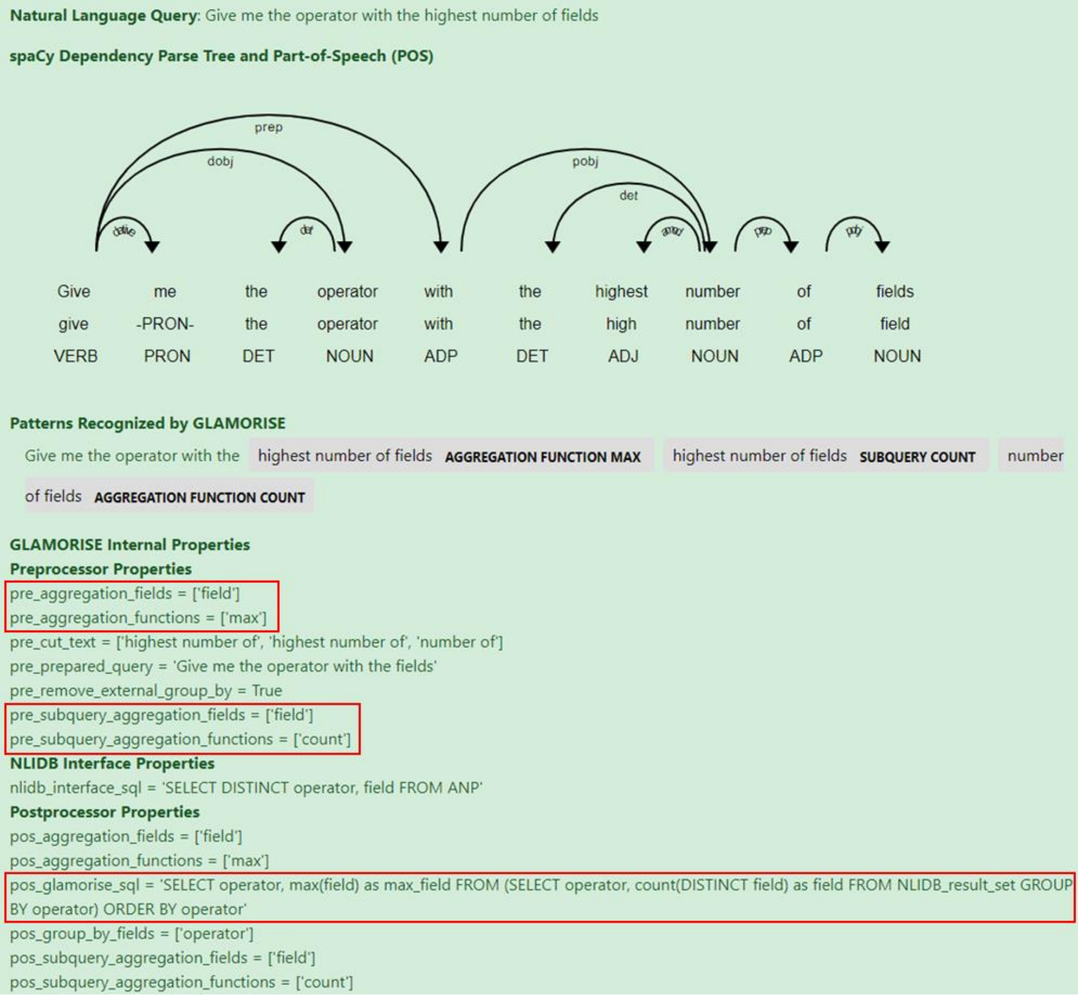
Postprocessor Properties

pos_aggregation_fields = ['oil_production']
pos_aggregation_functions = ['avg']
pos_glamorise_sql = 'SELECT avg(oil_production) as avg_oil_production FROM (SELECT sum(oil_production) as oil_production FROM NLIDB_result_set GROUP BY year)'
pos_subquery_aggregation_fields = ['oil_production']
pos_subquery_aggregation_functions = ['sum']

Figure 11 – Aggregations with timescale differences example

3.2.7. Subquery – Nested Aggregation Functions

A common pattern to be found in a NLQ is, for example, “highest number of” or “largest number of”. This kind of pattern is translated to an SQL query using two aggregation functions, in this case, a max function followed by a count function, which are nested due to the limitations of SQLite. One example is shown below:



Natural Language Query: Give me the operator with the highest number of fields

spaCy Dependency Parse Tree and Part-of-Speech (POS)

Give	me	the	operator	with	the	highest	number	of	fields
give	-PRON-	the	operator	with	the	high	number	of	field
VERB	PRON	DET	NOUN	ADP	DET	ADJ	NOUN	ADP	NOUN

Patterns Recognized by GLAMORISE

Give me the operator with the highest number of fields **AGGREGATION FUNCTION MAX** highest number of fields **SUBQUERY COUNT** number of fields **AGGREGATION FUNCTION COUNT**

GLAMORISE Internal Properties
Preprocessor Properties
pre_aggregation_fields = ['field']
pre_aggregation_functions = ['max']
pre_cut_text = ['highest number of', 'highest number of', 'number of']
pre_prepared_query = 'Give me the operator with the fields'
pre_remove_external_group_by = True
pre_subquery_aggregation_fields = ['field']
pre_subquery_aggregation_functions = ['count']
NLIDB Interface Properties
nlidb_interface_sql = 'SELECT DISTINCT operator, field FROM ANP'
Postprocessor Properties
pos_aggregation_fields = ['field']
pos_aggregation_functions = ['max']
pos_glamorise_sql = 'SELECT operator, max(field) as max_field FROM (SELECT operator, count(DISTINCT field) as field FROM NLIDB_result_set GROUP BY operator) ORDER BY operator'
pos_group_by_fields = ['operator']
pos_subquery_aggregation_fields = ['field']
pos_subquery_aggregation_functions = ['count']

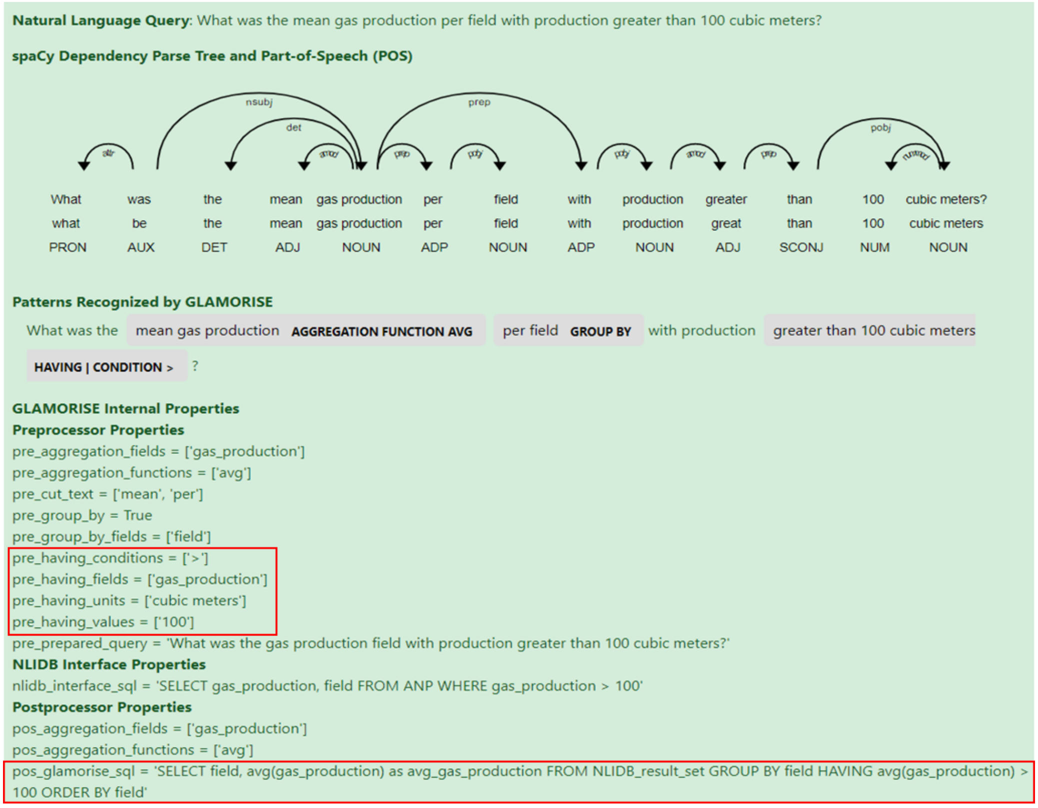
Figure 12 – Nested Aggregation Functions example

3.2.8. Ellipsis

Ellipsis is a broad subject and difficult to treat. In this dissertation, we tackle two types that are common in NLQs with aggregations. One of them is when GLAMORISE is faced with the use of a max, min or avg aggregation function over a string field. In this case, it realizes that some term, such as "number of", is implied in the sentence and solves the question by synthesizing a nested aggregation function subquery, as in the previous section. In this subquery the count aggregation function is first applied to the string field and its result is returned to the outer query to apply the max, min or avg function. The other case that is dealt with is when we have a having clause and the aggregation field and function is implied by the context of the rest of the sentence, as illustrated in Figure 13.

3.2.9. Having clause

Some NLQ patterns are translated into having clauses with conditions. For example, in the sentence "What was the mean gas production per field with production greater than 100 cubic meters?", the keywords "greater than" can be configured to capture a having clause. Figure 13 illustrates the interpretation of this sentence.



Natural Language Query: What was the mean gas production per field with production greater than 100 cubic meters?

spaCy Dependency Parse Tree and Part-of-Speech (POS)

What	was	the	mean	gas production	per	field	with	production	greater	than	100	cubic meters?
what	be	the	mean	gas production	per	field	with	production	great	than	100	cubic meters
PRON	AUX	DET	ADJ	NOUN	ADP	NOUN	ADP	NOUN	ADJ	SCONJ	NUM	NOUN

Patterns Recognized by GLAMORISE

What was the mean gas production **AGGREGATION FUNCTION AVG** per field **GROUP BY** with production greater than 100 cubic meters **HAVING | CONDITION >** ?

GLAMORISE Internal Properties
Preprocessor Properties

```
pre_aggregation_fields = ['gas_production']
pre_aggregation_functions = ['avg']
pre_cut_text = ['mean', 'per']
pre_group_by = True
pre_group_by_fields = ['field']
pre_having_conditions = ['>']
pre_having_fields = ['gas_production']
pre_having_units = ['cubic meters']
pre_having_values = ['100']
pre_prepared_query = 'What was the gas production field with production greater than 100 cubic meters?'
```

NLIDB Interface Properties

```
nlidb_interface_sql = 'SELECT gas_production, field FROM ANP WHERE gas_production > 100'
```

Postprocessor Properties

```
pos_aggregation_fields = ['gas_production']
pos_aggregation_functions = ['avg']
pos_glamorise_sql = 'SELECT field, avg(gas_production) as avg_gas_production FROM NLIDB_result_set GROUP BY field HAVING avg(gas_production) > 100 ORDER BY field'
```

Figure 13 – Having clause example

3.2.10. Limitations

It was decided not to deal with qualitative queries. Aggregation functions can be thought of as being of two types. *Quantitative aggregation functions* have a direct mapping to aggregation functions, such as *max*, *min*, *avg*, *count*, *sum*, and *qualitative aggregation functions*, such as *good*, *bad*, *high*, *low*, etc. , as discussed in [Abhijeet Gupta 2013; Gupta and Sangal 2013]. Databases do not handle qualitative aggregations natively as there is no direct mapping to aggregation functions. The problem we identified in dealing with qualitative aggregation functions is the lack of standardization: what is good or near for one person is not good or near for another.

In addition, as stated earlier, it is outside the scope of this work an advanced treatment of units of measures, such as conversions, all the nuances involving ellipsis, and also all possible cases involving subqueries, in addition to those treated in this section. We only dealt with a few cases involving subqueries and only on two levels. One type of subquery not covered, but common in natural language, is when the outer query has no aggregation and the nested query has aggregation, such as "Give me the fields that produce more oil than the average production of all fields ".

3.3. Technology

The GLAMORISE prototype was implemented in Python[5] with the help of the spaCy[6] library, to handle natural language processing. Additionally, it used regular expressions (regex) to identify some patterns that did not depend on the parse tree.

spaCy [Honnibal and Montani 2017] is an open-source software library for advanced natural language processing, written in Python and Cython[7]. Cython itself is a superset of Python and designed to give C-like performance. Using this combination, spaCy delivers good performance. It features convolutional neural network models for Part-Of-Speech (POS)-tagging, parse tree [Honnibal and Johnson 2015], text

[5] https://www.python.org/
[6] https://spacy.io/
[7] https://cython.org/

categorization and named-entity recognition (NER). In our work, the spaCy functionalities of POS-tagging and parse tree are used.

The result is presented to the user with the help of a Pandas Dataframe. Pandas[8] is an open-source library which provides easy-to-use high-performance data structures and data analysis tools for the program language.

GLAMORISE uses SQLite as its internal database. SQLite[9] is a relational database management system (RDBMS) written in C. Contrasting with many other database management systems, SQLite is not a client–server database engine. Rather, it is embedded into the end program. It implements most of the SQL standard, generally following the PostgreSQL syntax.

The Web version was developed in Flask[10] and used Bootstrap[11] to make it more visually pleasing.

Git[12] and Github[13] were used as version controller and version control repository, respectively.

3.4. Implementation

The current code consists of 8 classes. The main class is the **Glamorise** class, which is responsible for all independent implementation of the connected NLIDB. Additionally, this class

[8] https://pandas.pydata.org/
[9] https://www.sqlite.org/
[10] https://palletsprojects.com/p/flask/
[11] https://getbootstrap.com/
[12] https://git-scm.com/
[13] https://github.com/

includes the **Preprocessor** and **Postprocessor** layers, illustrated in Section 3.1.

The **GlamoriseNlidb** class is aware of the existence of the integrated NLIDB and implements the **Interface** layer, described in Section 3.1. This class has an instance of the class that integrates with the NLIDB. We implemented two classes of this type: **NlidbMock**, which is the class that implements the mock NLIDB used by the proof-of-concept; and the **NlidbNalir** class, responsible for the integration with the NLIDB NaLIR, used as the real NLIDB integrated in this work. A **NlidbDanke** class was also implemented by the DANKE team to perform the integration with this NLIDB. These three classes have a common ancestor, which is the **NlidbBase** class that holds the part of the code common to all NLIDB integration classes.

We still have two classes that are responsible for the steps that were inserted in the spaCy pipeline, as mentioned before. **CompoundMerger** is responsible for attributes with compound names and **UnitsOfMeasurementMerger** is responsible for the treatment of units of measurement. Both inherit from a class that has the necessary common code, called **Merger**.

Figure 14 shows the hierarchy of the GLAMORISE classes.

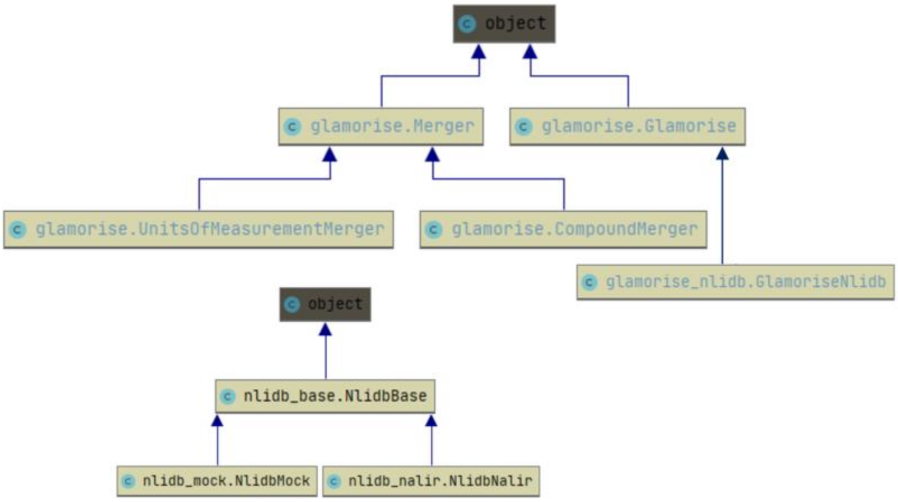


Figure 14 – GLAMORISE Classes Diagram

3.5. Repository and Reproducibility

All code, NLQs datasets and links to the databases can be found at https://github.com/novello/GLAMORISE for reuse or reproducibility of the experiments and results.

An online version of the ready-to-use experimental system can be found at http://glamorise.gruposantaisabel.com.br/

3.6. Configuration

3.6.1. Installation

The root path of the project on Github has a **README** file with all installation instructions.

3.6.2. Patterns Recognition and Configuration

GLAMORISE pattern recognition is implemented through rules that use reserved words followed by words that have a certain Part-Of-Speech (POS) or a certain relationship in the dependency tree. An intuitive way to describe how these rules work is to describe what are the parameters in the GLAMORISE configuration files.

The entire operation of GLAMORISE is governed by two JSON files. The main GLAMORISE configuration file defines how text patterns will be recognized and some visual parameters concerning how the results are exhibited, and the interface configuration file defines how the relationship with the integrated NLIDB will be governed, as detailed below:

Visual parameters:

- **show_dependency_parse_tree –** indicates whether the dependency parse tree is going to be shown in the result. It is a Boolean parameter.
- **show_recognized_patterns –** indicates whether the patterns that are recognized by GLAMORISE will be shown in the result in a visual way. It is a Boolean parameter.
- **total_row –** indicates whether the Pandas dataframe with the result set is shown in the result with a total row line. It is a Boolean parameter.
- **debug –** indicates whether the main internal properties of GLAMORISE layers (**Preprocessor**, **Interface** and

Postprocessor) are going to be shown in the result. It is a Boolean parameter.

Pattern parameters:

- **noun_lemmatization –** indicates whether the fields found will be lemmatized or not. The main idea is to be able to save plural nouns in the singular format, so that the logic of the system has to deal only with singular nouns. It is a Boolean parameter.
- **count_with_distinct** – indicates whether DISTINCT is going to be applied in a field when using a count aggregation function. The idea behind this is that, depending on how the data is stored in the database or how the result is given by the integrated NLIDB, there can be repetition of values for the tuples returned to a specific field and semantically, when it is asked how many items a field has, the intention behind this question is to know the number uniquely, without the repeated values. It is a Boolean parameter.
- **pre_before_replace_text** – indicates some terms that must be replaced in the NLQ in order to facilitate the match and interpretation of the NLQ. In the case of this option, the substitution is made before the NLQ is delivered for the interpretation of GLAMORISE. Usually this option is used in order to adjust some match problem in the integrated NLIDB. We do not recommend the use of

this functionality, we believe that the best way is to improve the way the match is made in the integrated NLIDB, use this option only as a workaround while this is not done. We also do not recommend using this feature for the same reasons as above.

- **pre_after_replace_text** – The same as the previous one, except that in this case the NLQ is delivered in its original form to GLAMORISE and this replacement only happens after the interpretation of GLAMORISE. It affects only the prepared NLQ that will be sent to the integrated NLIDB.
- **units_of_measurement –** is a list of strings with all units of measure that could appear in the domain of the specific database.
- **compound_pattern_dep** and **compound_pattern_of –** these parameters define how the compound names will be identified by the step that is added by GLAMORISE in the spaCy processing pipeline. It is a JSON object and the format is the rule-based Matcher component of spaCy, whether we define rules involving fixed text, Part-Of-Speech (POS) and relation between the word tokens in the dependency tree.
- **default_pattern** - this is the parameter used by most of the subsequent parameters to identify textual patterns that define an aggregation. This parameter defines the standard way in which a field is found. It also uses the

SpaCy Matcher component for this, and the factory-defined way is that a key field for GLAMORISE can be found after a keyword, defined in the subsequent rules. After this keyword can come an optional adverb (**{"POS": "ADV", "OP": "*"}**), an optional adjective (**{"POS": "ADJ", "OP": "*"}**) and finally a noun (**{"POS": "NOUN", "LOWER": {"NOT_IN": ["number"]}}**), which is the field to be identified. Recalling that this noun can be simple or compound, considering that the previous step of identifying compound nouns has already been undertaken and was responsible for conjoing them into a single noun in the dependency tree. The most attentive reader will notice the following part in the rule (**"LOWER": {"NOT_IN": ["number"]}**). The reason for this is that "number" and "number of" are reserved words that are often used to mean a counting function and the word "number" is a noun, so it should not be considered for field match purposes. This can lead to a problem if the field in the database has the word "number" in its name, in this case the person in charge of configuring GLAMORISE within the database needs to choose between the tradeoff of having a field with the word "number" in the database or use it as a keyword for the count function and adjust the rules accordingly.

- **config_glamorise –** this parameter is responsible for all the other pattern configurations involving the declaration of keywords, the expected patterns after them, and which triggers of aggregation clauses, aggregation functions, group by and conditions of the having clause each keyword triggers. Each set of rules is a JSON object within this parameter, which is freely named, and will be described later. Each of these objects are made up of other object parameters that define how that pattern is recognized. These parameters will be detailed later. Those with the prefix "pre_" are because they have direct mapping with GLAMORISE Preprocessor layer properties.
    - **reserved_words –** this a string list that contains the keywords that will be used as a trigger for that pattern recognizer;
    - **pre_having_conditions –** when the pattern to be recognized is a having clause, this parameter is used to specify a having condition for each keyword, so it is a string list with the same size of the **reserved_words** parameter and contains strings related to the conditions: ">", ">", "<", "=", ">=" and "<=".
    - **specific_pattern –** works in the same way as the default pattern parameter discussed previously

and serves, specifically, to override the behavior of this parameter for a specific rule.

- **pre_cut_text –** indicates whether the keyword must be cut or not from the NLQ before sending it to the integrated NLIDB. The purpose of cutting a keyword is to prevent the conventional NLIDB from being confused by the presence of an aggregation element. It is a Boolean parameter.
- **pre_group_by** – indicates whether the keyword triggers a group by condition. In this case, the field identified is going to be inserted in the SELECT, GROUP BY and ORDER BY clauses. It is a Boolean parameter.
- **pre_aggregation_functions** - when the pattern to be recognized is an aggregation function, this parameter is used to specify an aggregation function for each keyword, so it is just a string if the same function is applied to all **reserved_words** or a string list with the same size of the **reserved_words** parameter and containing strings related to the aggregation functions of each keyword. The functions usually are: "count", "max", "min", "avg" and "sum".
- **pre_subquery_replace_text –** This parameter is used by the time-scale feature which is resolved

through the subquery functionality. In this case it is usually necessary to replace the original word, which is an adjective for a noun, since it is generally easier for the integrated NLIDB to match the related field in the form of a noun. Example "yearly" being replaced by "year".

- **use_replace_text_as_group_by –** This parameter is also used by the time-scale feature. It is a Boolean parameter. It must be set as true when used in a timescale subquery pattern, but it must be set to false when used with the other type of subquery (nested aggregation function) dealt with.
- **pre_subquery_aggregation_functions –** works the same as the **pre_aggregation_functions** parameter, but in this case the aggregation functions will be used in the subquery.
- **remove_external_group_by –** this is a trick to get the right answer when using the nested aggregation function, in this case it must be set to true, and when using the timescale it must be set to false.

The second file has parameters that define the relationship between GLAMORISE and the integrated NLIDB, as described below:

- **nlidb_field_synonym –** This is the main configuration parameter for this file. In this parameter, the matching of several strings with the respective fields in the database is configured. The configuration can be thought of as a table, where the words that must match the database fields are in the first column, separated by underscore, and in the second column the fields in the database itself in the format "table.column". As the strings already arrive at this stage with lemmatized and compound names, it is not necessary to make variations of the same word as singular and plural or in the case of compound names it is not necessary to register the "noun of noun" format, just the "noun noun" format. For example, just register “oil_production”, which the system translates if it is stored as “production_of_oil” in the database. If a word that is not registered in this list appears, the spaCy similarity function that uses word vectors will be used to search for which word has the greatest similarity with the unknown word and so choose the best guess of field to match.

- **nlidb_nlq_translate_fields –** This parameter must be avoided being used. Its idea is to replace, in the NLQ to be sent to the integrated NLIDB, the words referring to the fields with the database fields themselves. This property should only be used if the integrated NLIDB match is giving awfully bad results, since the NLQ sent to NLIDB will undergo significant changes in terms of natural language semantics.

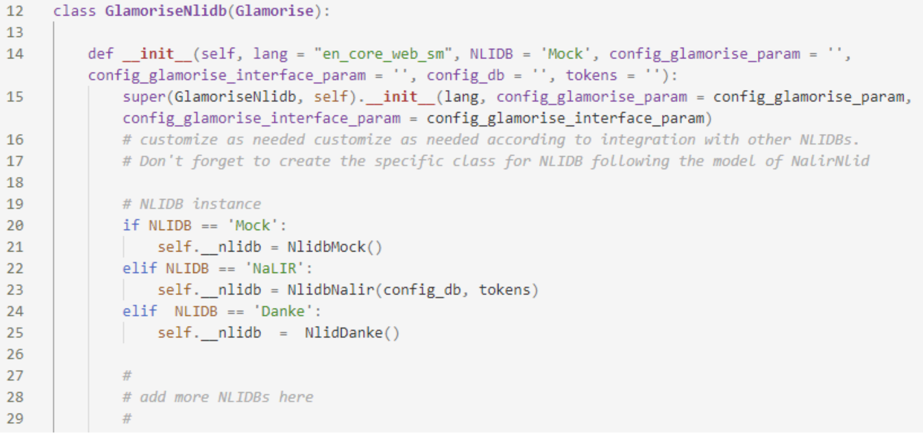
- **nlidb_attempt_level –** This parameter is dependent on the way the integration with the NLIDB was implemented and more about that will be said in Section 3.6.3. In the NLIDB Mock, only 1 attempt level is implemented and in the NaLIR 3 attempt levels are implemented. The parameter must be set to specify to which level it must reach during execution.
- **nlidb_aggregation –** This parameter must also be avoided. The idea of this parameter is that every string field is placed in the GROUP BY, and in every numeric field the sum function is applied to the result set returned from NLIDB before GLAMORISE performs its aggregation. This field gives more flexibility if the objective is to use the system without making explicit the use of all the aggregation keywords and also as a facilitator if the database storage format is leading to many incorrect answers. We advise against using this feature because we believe that the best way to solve this problem is to prepare the database format to match the answers that are expected and to explicitly use aggregation keywords, otherwise questions that do not have keywords of aggregation will always have aggregated responses, even if this is not the objective.
- **nlidb_aggregation_exceptions –** This parameter specifies which numeric fields should not be submitted to the sum

function if the previous parameter is activated. For example, if fields related to dates are stored in numeric format, it does not make sense that they are submitted to the sum function. It also uses the format "table.column".

3.6.3. Integration of a new NLIDB with GLAMORISE

One of the steps that needs to be performed for GLAMORISE to work correctly is the integration with a conventional NLIDB capable of processing a NLQ without the aggregation elements. Since GLAMORISE is an open-source experimental software, the most flexible way to allow this is by directly changing the source code. This section describes the necessary steps to integrate GLAMORISE with another NLIDB.

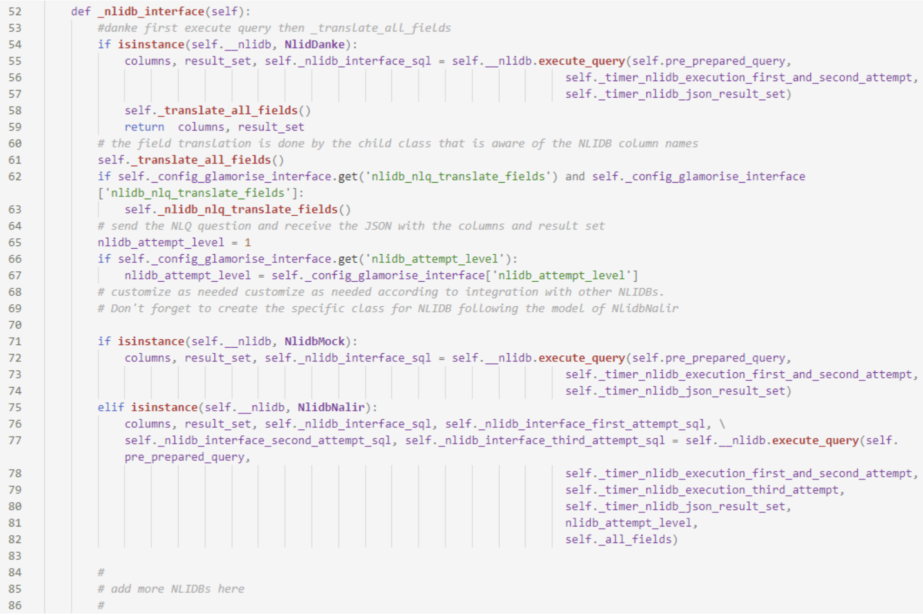
Initially, in the **GlamoriseNlidb** class, it is necessary to prepare the constructor (**__init__**) to instantiate the new type of NLIDB, which is done by adding one more **elif** with the string equivalent to the name of the NLIDB to be passed as a parameter to the class constructor and creating the new instance of the NLIDB. The code snippet for this part is shown below:



```
class GlamoriseNlidb(Glamorise):

    def __init__(self, lang = "en_core_web_sm", NLIDB = 'Mock', config_glamorise_param = '',
    config_glamorise_interface_param = '', config_db = '', tokens = ''):
        super(GlamoriseNlidb, self).__init__(lang, config_glamorise_param = config_glamorise_param,
        config_glamorise_interface_param = config_glamorise_interface_param)
        # customize as needed customize as needed according to integration with other NLIDBs.
        # Don't forget to create the specific class for NLIDB following the model of NalirNlid

        # NLIDB instance
        if NLIDB == 'Mock':
            self.__nlidb = NlidbMock()
        elif NLIDB == 'NaLIR':
            self.__nlidb = NlidbNalir(config_db, tokens)
        elif  NLIDB == 'Danke':
            self.__nlidb  =  NlidDanke()

        #
        # add more NLIDBs here
        #
```

Listing 12 – Code snippet - class GlamoriseNlidb, method __init__

After that, it is necessary, in the **_nlidb_interface** method, to add another **elif** to the **execute_query** method of the NLIDB interface class instance, which is responsible for processing the NLQ without aggregation in the integrated NLIDB. The code snippet is below:



```
    def _nlidb_interface(self):
        #danke first execute query then _translate_all_fields
        if isinstance(self.__nlidb, NlidDanke):
            columns, result_set, self._nlidb_interface_sql = self.__nlidb.execute_query(self.pre_prepared_query,
                                                                  self._timer_nlidb_execution_first_and_second_attempt,
                                                                  self._timer_nlidb_json_result_set)
            self._translate_all_fields()
            return  columns, result_set
        # the field translation is done by the child class that is aware of the NLIDB column names
        self._translate_all_fields()
        if self._config_glamorise_interface.get('nlidb_nlq_translate_fields') and self._config_glamorise_interface
        ['nlidb_nlq_translate_fields']:
            self._nlidb_nlq_translate_fields()
        # send the NLQ question and receive the JSON with the columns and result set
        nlidb_attempt_level = 1
        if self._config_glamorise_interface.get('nlidb_attempt_level'):
            nlidb_attempt_level = self._config_glamorise_interface['nlidb_attempt_level']
        # customize as needed customize as needed according to integration with other NLIDBs.
        # Don't forget to create the specific class for NLIDB following the model of NlidbNalir

        if isinstance(self.__nlidb, NlidbMock):
            columns, result_set, self._nlidb_interface_sql = self.__nlidb.execute_query(self.pre_prepared_query,
                                                                  self._timer_nlidb_execution_first_and_second_attempt,
                                                                  self._timer_nlidb_json_result_set)
        elif isinstance(self.__nlidb, NlidbNalir):
            columns, result_set, self._nlidb_interface_sql, self._nlidb_interface_first_attempt_sql, \
            self._nlidb_interface_second_attempt_sql, self._nlidb_interface_third_attempt_sql = self.__nlidb.execute_query(self.
            pre_prepared_query,
                                                                  self._timer_nlidb_execution_first_and_second_attempt,
                                                                  self._timer_nlidb_execution_third_attempt,
                                                                  self._timer_nlidb_json_result_set,
                                                                  nlidb_attempt_level,
                                                                  self._all_fields)

        #
        # add more NLIDBs here
        #
```

Listing 13 – Code snippet - class GlamoriseNlidb, method _nlidb_interface

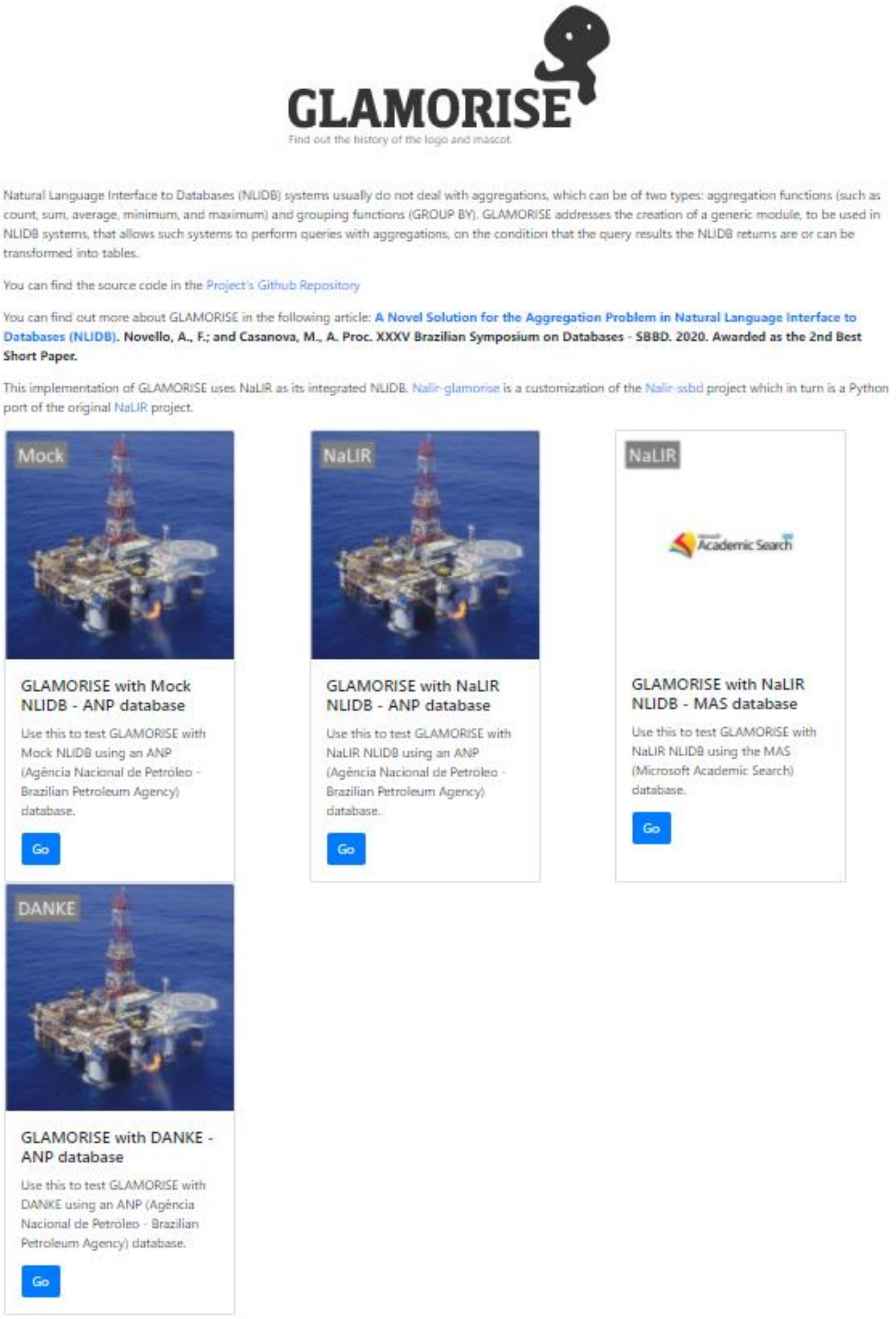
The final step is the implementation of the interface class with the integrated NLIDB. This class must have at least one public method, **execute_query**, which, as mentioned previously, is responsible for executing NLQ without aggregation in the integrated NLIDB. In the source code made available on Github, it is possible to check the implementation of this method in 3 different classes: **NlidbMock**, **NlidbNalir** and **NlidbDanke**.

3.7. User Interface

3.7.1. Web Interface

For convenience, a simple Web interface was developed to test GLAMORISE. Its code is located in the web_interface folder in the Github repository and an online version is also available at https://glamorise.gruposantaisabel.com.br/.

The first screen shows which options of integrated NLIDBs and which datasets are available. Each option represents a duo of NLIDB and dataset, since a previous configuration is required for each dataset. The following figure shows the options available:



Natural Language Interface to Databases (NLIDB) systems usually do not deal with aggregations, which can be of two types: aggregation functions (such as count, sum, average, minimum, and maximum) and grouping functions (GROUP BY). GLAMORISE addresses the creation of a generic module, to be used in NLIDB systems, that allows such systems to perform queries with aggregations, on the condition that the query results the NLIDB returns are or can be transformed into tables.

You can find the source code in the Project's Github Repository

You can find out more about GLAMORISE in the following article: **A Novel Solution for the Aggregation Problem in Natural Language Interface to Databases (NLIDB). Novello, A., F.; and Casanova, M., A. Proc. XXXV Brazilian Symposium on Databases - SBBD. 2020. Awarded as the 2nd Best Short Paper.**

This implementation of GLAMORISE uses NaLIR as its integrated NLIDB. Nalir-glamorise is a customization of the Nalir-ssbd project which in turn is a Python port of the original NaLIR project.

GLAMORISE with Mock NLIDB - ANP database

Use this to test GLAMORISE with Mock NLIDB using an ANP (Agência Nacional de Petróleo - Brazilian Petroleum Agency) database.

GLAMORISE with NaLIR NLIDB - ANP database

Use this to test GLAMORISE with NaLIR NLIDB using an ANP (Agência Nacional de Petróleo - Brazilian Petroleum Agency) database.

GLAMORISE with NaLIR NLIDB - MAS database

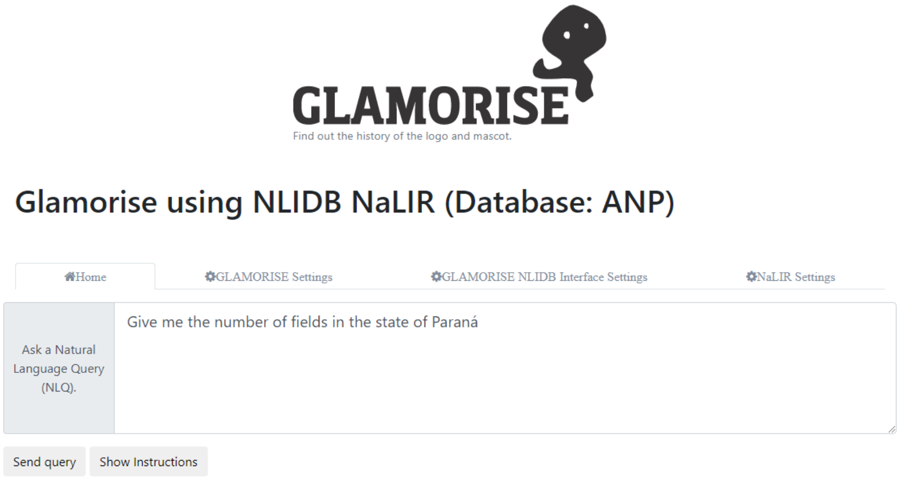
Use this to test GLAMORISE with NaLIR NLIDB using the MAS (Microsoft Academic Search) database.

GLAMORISE with DANKE - ANP database

Use this to test GLAMORISE with DANKE using an ANP (Agência Nacional de Petróleo - Brazilian Petroleum Agency) database.

Figure 15 – Web Interface – Main Screen

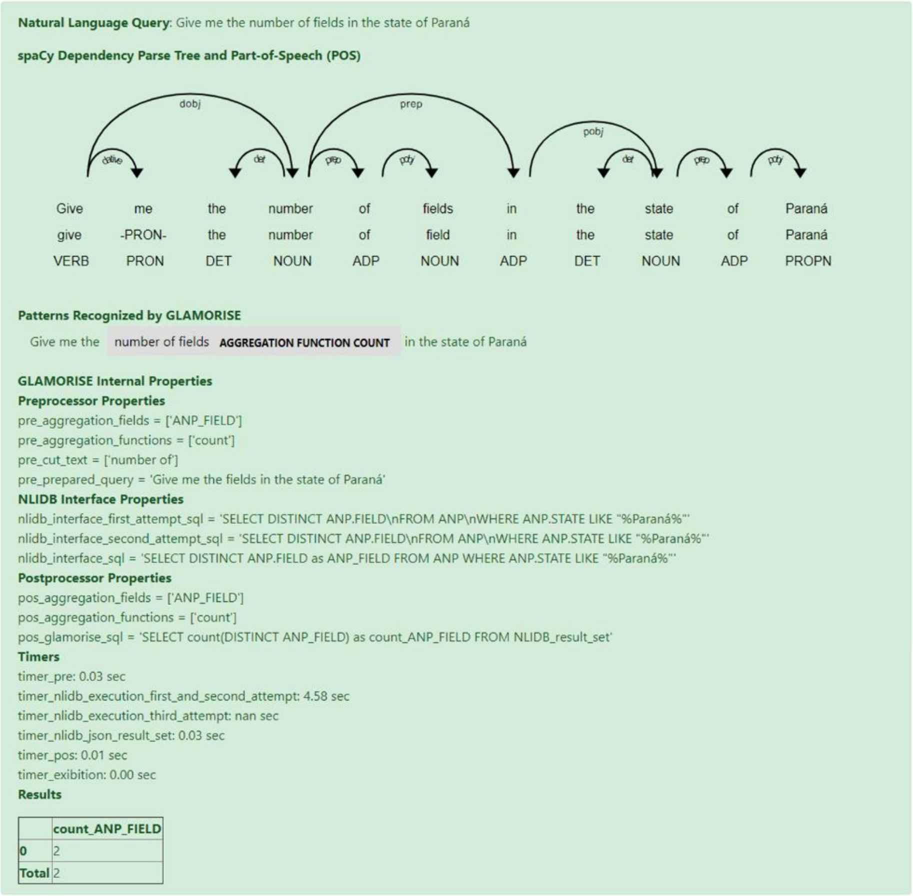
On the next screen we can see, when it is loaded, a field where the NLQ can be written. To process the NLQ, the **Send Query** button must be pressed. If the user is unsure how to ask a question, he can choose the **Show Instructions** option.


GLAMORISE
Find out the history of the logo and mascot.
Glamorise using NLIDB NaLIR (Database: ANP)
Home
GLAMORISE Settings
GLAMORISE NLIDB Interface Settings
NaLIR Settings
Ask a Natural Language Query (NLQ).
Send query
Show Instructions

Give me the number of fields in the state of Paraná

Figure 16 – Web Interface – NaLIR Integration – NLQ text area

The answer to the NLQ appears in a green rectangle below. If the default configurations are used, initially the spaCy dependency tree is shown, then the patterns recognized in the sentence by GLAMORISE, followed by the internal GLAMORISE variables, and finally the answer, the result set in table format.



Natural Language Query: Give me the number of fields in the state of Paraná

spaCy Dependency Parse Tree and Part-of-Speech (POS)

Give	me	the	number	of	fields	in	the	state	of	Paraná
give	-PRON-	the	number	of	field	in	the	state	of	Paraná
VERB	PRON	DET	NOUN	ADP	NOUN	ADP	DET	NOUN	ADP	PROPN

Patterns Recognized by GLAMORISE

Give me the number of fields **AGGREGATION FUNCTION COUNT** in the state of Paraná

GLAMORISE Internal Properties

Preprocessor Properties

```
pre_aggregation_fields = ['ANP_FIELD']
pre_aggregation_functions = ['count']
pre_cut_text = ['number of']
pre_prepared_query = 'Give me the fields in the state of Paraná'
```

NLIDB Interface Properties

```
nlidb_interface_first_attempt_sql = 'SELECT DISTINCT ANP.FIELD\nFROM ANP\nWHERE ANP.STATE LIKE "%Paraná%"'
nlidb_interface_second_attempt_sql = 'SELECT DISTINCT ANP.FIELD\nFROM ANP\nWHERE ANP.STATE LIKE "%Paraná%"'
nlidb_interface_sql = 'SELECT DISTINCT ANP.FIELD as ANP_FIELD FROM ANP WHERE ANP.STATE LIKE "%Paraná%"'
```

Postprocessor Properties

```
pos_aggregation_fields = ['ANP_FIELD']
pos_aggregation_functions = ['count']
pos_glamorise_sql = 'SELECT count(DISTINCT ANP_FIELD) as count_ANP_FIELD FROM NLIDB_result_set'
```

Timers

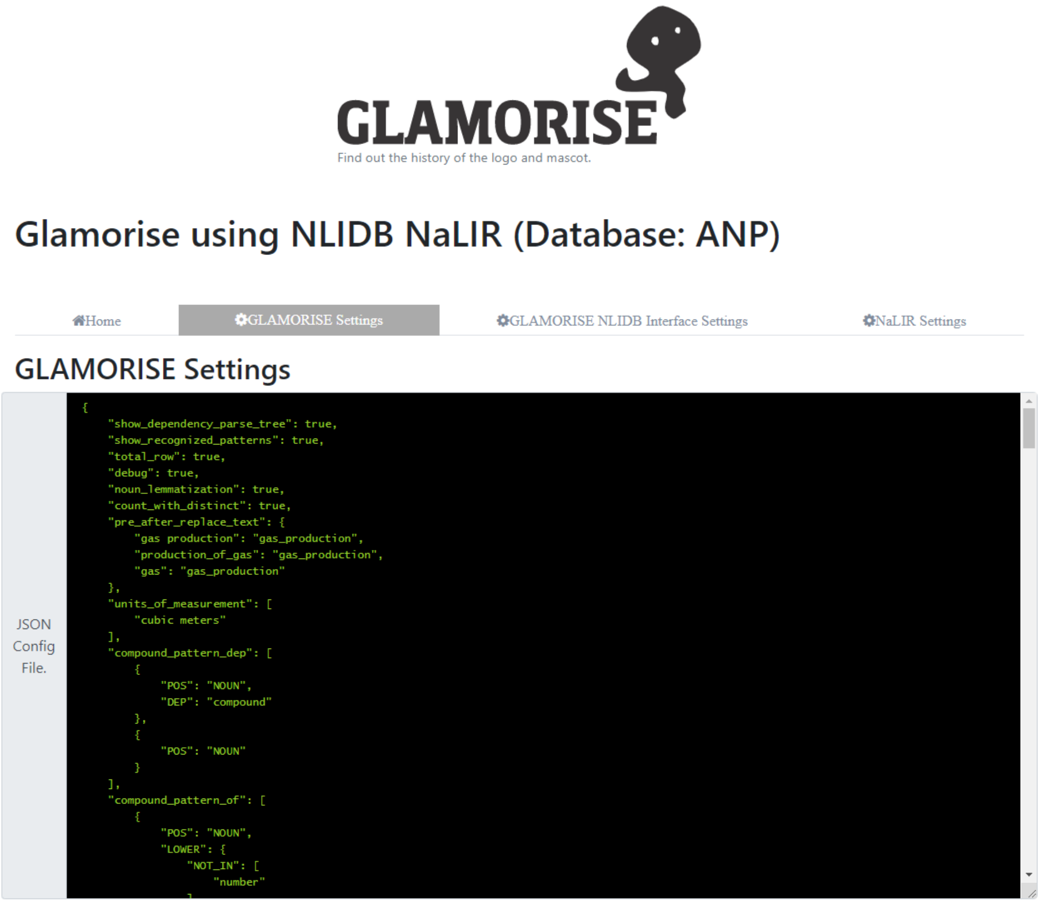
```
timer_pre: 0.03 sec
timer_nlidb_execution_first_and_second_attempt: 4.58 sec
timer_nlidb_execution_third_attempt: nan sec
timer_nlidb_json_result_set: 0.03 sec
timer_pos: 0.01 sec
timer_exibition: 0.00 sec
```

Results

	count_ANP_FIELD
0	2
Total	2

Figure 17 – Web Interface – NaLIR Integration – Result

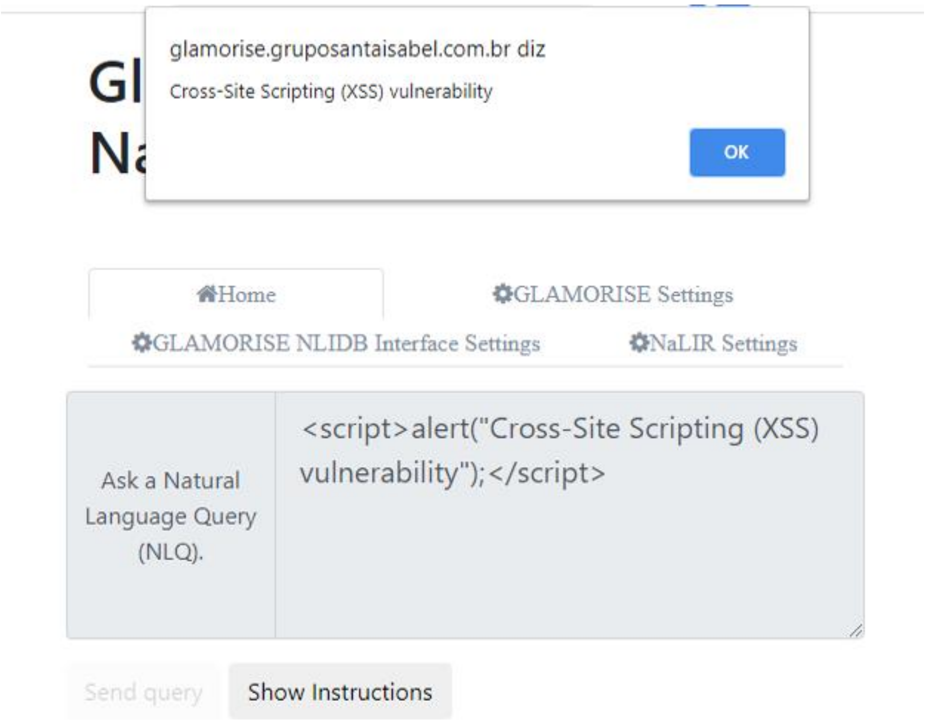
As we can see, some tabs are available. Each tab represents a configuration file, either from GLAMORISE or from the integrated NLIDB. The first two files refer to GLAMORISE, the third onwards, if any, refer to the integrated NLIDB. The GLAMORISE files have already been presented in Section 3.6.2 and in the example image below we can see that it is NLIDB NaLIR and that it has a configuration file, which will be described in Section 4.2.1.


GLAMORISE
Find out the history of the logo and mascot.
Glamorise using NLIDB NaLIR (Database: ANP)
Home
GLAMORISE Settings
GLAMORISE NLIDB Interface Settings
NaLIR Settings
GLAMORISE Settings
JSON Config File.

```
{
    "show_dependency_parse_tree": true,
    "show_recognized_patterns": true,
    "total_row": true,
    "debug": true,
    "noun_lemmatization": true,
    "count_with_distinct": true,
    "pre_after_replace_text": {
        "gas production": "gas_production",
        "production_of_gas": "gas_production",
        "gas": "gas_production"
    },
    "units_of_measurement": [
        "cubic meters"
    ],
    "compound_pattern_dep": [
        {
            "POS": "NOUN",
            "DEP": "compound"
        },
        {
            "POS": "NOUN"
        }
    ],
    "compound_pattern_of": [
        {
            "POS": "NOUN",
            "LOWER": {
                "NOT_IN": [
                    "number"
                ]
```

Figure 18 – Web Interface – NaLIR Integration – Configuration Files

GLAMORISE does not accept in its NLQs the use of the symbols: ">" and "<". If the user wants to express these conditions, he needs to write conditions verbatim such as "greater than" or "less than". One reason is that textual expressions have more semantic value than symbols in an NLQ. The other objective of this limitation is to protect the Web interface against Cross-Site Scripting (XSS), which is when there is injection of Javascript on the client side. Below an image is shown of the type of XSS that could occur, a Javascript code could be passed by the client through the NLQ to GLAMORISE and if it was able to accept this type of symbol (“<” or “>”), the Javascript code would be returned encapsulated as part of the answer and would be executed:



glamorise.gruposantaisabel.com.br diz

Cross-Site Scripting (XSS) vulnerability

Ask a Natural Language Query (NLQ).

<script>alert("Cross-Site Scripting (XSS) vulnerability");</script>

Figure 19 – Web Interface – Cross-Site Scripting (XSS)

3.7.2. Terminal Script Interface

The user can also test GLAMORISE using the terminal. There are several scripts to help you with this task. You can use the following scripts to test GLAMORISE with a single NLQ. You can check out the name of the NLIDB and dataset by the script file name:

- main_mock_anp_single_nlq.py
- main_nalir_anp_single_nlq.py

- main_nalir_mas_single_nlq.py
- main_danke_anp_single_nlq.py

In order to test, it is necessary to edit the file and change the NLQ to the one of your preference.

If you want to test a batch of NLQs, you can use the following scripts:

- main_mock_anp.py
- main_nalir_anp.py
- main_nalir_mas.py
- main_danke_anp.py

The ANP dataset NLQs are in the file **./nlqs/anp.nlqs.txt** and the MAS dataset NLQs are in the file **./nlqs/nalir_mas_aggregation_subset.nlqs.txt.**

4. Experiments

This section describes the experiments conducted to test GLAMORISE.

4.1. A Proof-of-Concept with a Mock NLIDB

To test the performance of GLAMORISE, a mock NLIDB was implemented to process the set of testing questions presented in Table 2. Note that there are questions with completely different phrasings. The tests first confirmed GLAMORISE correctly preprocessed the questions, removing or substituting words (aggregation elements) as necessary. Second, the tests confirmed that GLAMORISE correctly generated SQL queries with aggregation.

Finally, GLAMORISE correctly answered all 22 questions in Table 2.

ID	NLQ
Q1	What was the production of oil in the state of Rio de Janeiro?
Q2	What was the average monthly production of oil in the state of Rio de Janeiro?
Q3	What was the average yearly production of oil in the state of Alagoas?
Q4	How many fields are there in Paraná?
Q5	What was the maximum production of oil in the state of Ceará per field?
Q6	What was the minimum gas production in the state of São Paulo per basin?
Q7	What was the average monthly oil production by the operator Petrobrás?
Q8	What was the mean yearly gas production per field?
Q9	What was the mean gas production per month per field?
Q10	What was the per month mean gas production per field?
Q11	What was the per field mean gas production per month?
Q12	What was the mean monthly petroleum production by field in the state of Rio de Janeiro?
Q13	What was the mean yearly petroleum production by field by Rio de Janeiro?
Q14	What was the mean gas production per field with production greater than 100 cubic meters?
Q15	What was the mean gas production per basin with production less than 1000 cubic meters?
Q16	Which field produces the most oil per month?
Q17	Which basin has the highest yearly oil production?
Q18	Which federated state has the lowest gas production?
Q19	Which state of the federation has the lowest gas production?
Q20	What was the average yearly production of oil per field and state in the year 2015?
Q21	What was the average monthly production of oil per field in the state of Rio de Janeiro and year 2015?
Q22	Give me the operator with the highest number of fields.

Table 2 – ANP NLQs

4.2. Real NLIDB Integration

The next step was to integrate GLAMORISE with real NLIDBs. The two NLIDBs described below were chosen.

4.2.1. NaLIR

NaLIR [Li and Jagadish, H. V. 2014] was the first NLIDB to be integrated with GLAMORISE. As mentioned in Section 1.3, in a comparative survey containing 26 NLIDBs [Affolter et al. 2019], NaLIR performed best, as shown in Figure 3. Furthermore, NaLIR is capable of answering aggregation questions. So, as to probe GLAMORISE’s ability to deal with aggregation, the aggregation layer was removed from NaLIR, leaving GLAMORISE to address this issue, while NaLIR addressed the remainder of the issues involved in an NLIDB.

To facilitate the initial integration, the original Java implementation of NaLIR[14] was not adopted. Instead, a port to Python[15] [16] was employed, which is the same language used to develop GLAMORISE. To remove the aggregation ability of NaLIR, the original source code was slightly modified, generating another project also available on Github[17].

Another important consideration is that the original NaLIR has a layer, called Interactive Communicator, which the user interacts with to refine the answer. The implementation used suppressed this layer, which is consistent with the strategy implemented with GLAMORISE.

The integration of NaLIR with GLAMORISE was done via source code and following the precepts described in Section 3.6.3.

4.2.1.1. Configuration

Like GLAMORISE, NaLIR also has its own configuration files: two JSON files and one in XML. The JSON files are responsible for reassembling the database schema, although this information could be extracted directly from the database catalog. The default for the first file name is the database name followed by Edges (in our experiments the files are **masEdges.json** and **anpEdges.json**). This file indicates the connections between the tables and is composed by a JSON array of objects, each entry is one JSON object composed of four properties representing the connection between a primary key and a foreign key. These properties are described below:

- **foreignAttribute** – The table column name for the field that is the foreign key.
- **primaryAttribute**– The table column name for the field that is the primary key.
- **foreignRelation**– The name of the table that contains the foreign key.

[14] https://github.com/umich-dbgroup/NaLIR
[15] https://sbbd.org.br/2020/tutorial-1/
[16] https://github.com/pr3martins/nalir-sbbd
[17] https://github.com/novello/nalir-glamorise

- **primaryRelation** – The name of the table that contains the primary key.

The second JSON file is responsible for mapping the table fields that are visible to NaLIR. The default for the file name is the database name followed by Relations (in our experiments the files are **masRelations.json** and **anpRelations.json**). It is composed of a JSON array of objects, each entry is a JSON object composed of the following properties:

- **name** – the name of the table.
- **attributes** – is a JSON array of objects, each entry is a table column that should be visible to NaLIR:
    - **importance** – this attribute helps in the column / table match when any word in the NLQ generates a match for the table and it is necessary to know in which column the value should be searched for. Usually the values "important" or "primary" are used, the latter being the default of the table.
    - **name** – the name of the column.
    - **type** – the type of the column. Usually the values "pk", "fk", "text" or "number" are used
- **type** – the type of table. The values "entity" or "relationship" are used, the latter being for tables representing relationships.

The last file that is an XML and is responsible for designating the patterns recognized by NaLIR. The default file name is **tokens.xml**. The main tag is called <**types**>, below this the main tags are:

- **<CMT_V>** - command tokens and verbs, these patterns are used as a trigger to an NLQ
- **<OBT>** - order by token, these patterns are used to specify an order by clause
- **<FT>** - function token, these patterns area used to specify aggregation functions. We deliberately eschewed entries in this tag in our configuration to preclude NaLIR from performing this task and thereby assigning its performance to GLAMORISE.

- **<OT>** - operator token, these patterns area used to specify operator tokens like: “>”, “<”, “>=”, “<=” and “=”.
- **<QT>** - Qualifier token.
- **<NEG>** - Negative token.

Each of these tags has several <**phrase**> entries. One entry for each textual pattern to be recognized. Within each <**phrase**> tag we can have an <**example**> tag that exemplifies its use in an NLQ and is only descriptive. We also have special tags that can be used according to the parent tag as below:

- **<function>** inside **<FT>** - used to specify the aggregation function that should be used, but as previously mentioned, we do not use these entries in our configuration, so preventing NaLIR from performing this task so that GLAMORISE may do so.
- **<operator>** inside **<OT>** - used to specify the operator should be used.
- **<quantity>** inside **<QT>** - used to specify the quantity should be used. They could be: “all”, “each” and “any”.

4.2.1.2. Agência Nacional de Petróleo (ANP) [National Petroleum Agency] dataset results

In the scenario of GLAMORISE integrated with NaLIR, the same 22 proof-of-concept NLQs were presented using the ANP database as a baseline. NaLIR has a poor match with the gas production attribute, so we used a configuration property, that we do not recommend using, **pre_after_replace_text**, to improve this match. Ideally, NaLIR’s match ability should be improved, but this is outside the scope of this work and does not interfere with the functioning and evaluation of GLAMORISE.

GLAMORISE correctly answered 22 (100%), regarding the part under its responsibility and NaLIR correctly answered 18 (~82%) queries, regarding the part that was its responsibility, leading to a final result of 18 (~82%) correctly answered queries.

status/NLIDB	GLAMORISE	%	NaLIR	%	Final Result	%
success	22	100%	18	82%	18	82%
failure	0	0%	4	18%	4	18%
Total	**22**	**100%**	**22**	**100%**	**22**	**100%**

Table 3 – NaLIR results with ANP dataset

Most of NaLIR's errors were improper matches between one field and another, or it was simply unable to perform a particular match. In one NLQ (**Q5**), the “basin” field was incorrectly identified in the place of the “state” field, this happens because Ceará is also the name of a basin, which leads to an improper match. In another (**Q17**), it was unable to identify the “basin” field. In two other NLQs (**Q18 and Q19**), it was unable to identify the “state” field.

4.2.1.3. Microsoft Academic Search (MAS) dataset results

Microsoft Academic Search (MAS) is used by NaLIR as one of its databases. 194 NLQs[18] were originally created as a benchmark, but one of them is duplicated (‘return me the author who has the highest number of papers containing keyword "Relational Database".’). Of the 194 NLQs, 99 referred to questions with aggregation, but within these the linguistic and structural patterns recurred repeatedly. So, we chose 17 NLQs, presented in Table 4, that represent the universe of questions contained in the article, to determine/evaluate GLAMORISE’s performance.

ID	NLQ
Q1	return me the author in the "University of Michigan" in Databases area whose papers have more than 5000 total citations.
Q2	return me the author in the "University of Michigan" whose papers have the most total citations.
Q3	return me the author who has the most number of papers containing keyword "Relational Database".
Q4	return me the conference that has the most number of papers containing keyword "Relational Database".
Q5	return me the keyword, which have been contained by the most number of papers in PVLDB.
Q6	return me the number of authors who have cited the papers by "H. V. Jagadish".
Q7	return me the number of authors who have more than 10 papers containing keyword "Relational Database".
Q8	return me the number of citations of "Making database systems usable" in each year.
Q9	return me the number of conferences, which have more than 60 papers containing keyword "Relational Database".
Q10	return me the number of keywords, which have been contained by more than 10 papers of "H. V. Jagadish".
Q11	return me the number of keywords.
Q12	return me the number of papers after 2000 in "University of Michigan".
Q13	return me the number of papers published in PVLDB in each year.
Q14	return me the papers written by "H. V. Jagadish" and "Divesh Srivastava" with the most number of citations.
Q15	return me the total citations of all the papers in PVLDB.

[18] https://raw.githubusercontent.com/umich-dbgroup/NaLIR/master/mas_all.nlqs

Q16	return me the total citations of the papers containing keyword "Natural Language"
Q17	return me the total citations of papers in PVLDB before 2005.

Table 4 –MAS NLQs

The match of NaLIR with the word “paper” to designate a publication is very poor. Although it sometimes returns the column of the title of the publication correctly, the existence of the word “paper” generally leads it to erroneously believe that the title of the publication must contain the word paper. We are unaware as to whether it is a problem with the NaLIR Python version used or with the original version, but we believe it to be a problem of the ported version. As a result, we also used the **pre_after_replace_text** property. Of these queries, GLAMORISE correctly answered 17 NLQs (100%) regarding the part under its responsibility, and the NaLIR correctly answered 11 NLQs (~65%) queries, regarding the part that was its responsibility, leading to a final result of 11 NLQs (~65%) correctly answered.

status/NLIDB	GLAMORISE	%	NaLIR	%	Final Result	%
success	17	100%	11	65%	11	65%
failure	0	0%	6	35%	6	35%
Total	**17**	**100%**	**17**	**100%**	**17**	**100%**

Table 5 – NaLIR results with MAS dataset

Again, most of NaLIR's errors were related to its inability to perform a particular match, or improper matches between one field and another. In **Q1,** it mistakenly exchanged citations for references. In **Q7**, **Q9** and **Q10,** it mistakenly exchanged publications for references. In **Q6** and **Q14,** it was unable to join two instances of the author table. In **Q7** and **Q9,** it mistakenly concluded that "Relational Database" should be in the title of the publication and not in the keywords.

4.2.2. DANKE

The last experiment integrated GLAMORISE with an NLIDB natively incapable of handling aggregation. We also wanted to integrate GLAMORISE with an NLIDB that processed queries using RDF/SPARQL to prove that GLAMORISE works independently of the structure of the NLIDB database chosen, despite the fact that GLAMORISE uses a relational database (SQLite). DANKE [García 2020; Izquierdo et al. 2018, 2020; Torres Izquierdo et al. 2020] offered us these two opportunities. DANKE is an NLIDB of the Keyword Search (KwS) type and, as such, it is unable to answer questions with aggregation. DANKE operates with relational databases (synthesizing SQL queries) and RDF store (synthesizing SPARQL queries). It can deal with projections and selections and is able to create joins involving several tables. DANKE processes a keyword query without user intervention, consistent with the GLAMORISE strategy.

A keyword query is just a list of keywords that the user wishes to search for without the need of stop words and all the elements involved in a complete NLQ. DANKE permits the use of some reserved terms, such as "<", ">" (but not through the integration with GLAMORISE).

A response to a keyword query has a tabular format whose columns or column names contain the matches with the keywords.

The first step in using DANKE is to register the database, which is performed only once. The main tasks are to specify which columns have indexes and to add descriptions to the relation schemes and attributes. These descriptions provide the terms against which DANKE will match the keywords as well as the column values. The next step it to compile the database schema as an abstract schema, which is independent of whether it is relational database or an RDF store.

Once DANKE receives a keyword query, the first step it to match the keywords using the database and its schema. Then, an abstract query is created by the exploration of the schema and the keyword matches. Finally, a structured query (SQL or SPARQL) is compiled from the abstract query and executed.

The integration of GLAMORISE with DANKE was done by the DANKE team. Initially, the integration made with DANKE was in the black box model, that is, without access to its source code and was carried out via a Web service, considering that GLAMORISE is implemented in Python, whereas DANKE is in Java and the integration of the two via code would be costlier. On the other hand, this path substantially compromises performance and only serves to validate the concept of integrating GLAMORISE with a keyword search tool. Thereafter, the idea of the DANKE team is to port GLAMORISE to Java.

4.2.2.1. Agência Nacional de Petróleo - National Petroleum Agency (ANP) dataset results

The integration of GLAMORISE with DANKE produced excellent results. However, we made some adjustments before running the queries. States with compound names were enclosed in quotation marks, as DANKE does not present a good match when there are compound names. We had to use the **pre_after_replace_text** property again and also the **pre_before_replace_text**, which are not recommended, given that, when DANKE matches the month, it does not understand that it needs to return the month and year. The treatment of these issues needs to be improved in DANKE.

In general, DANKE performed faster than NaLIR, due to the keyword search technology used, and provides a more assertive result. Table 6 shows the results of the NLQs in Table 2. Of these queries, GLAMORISE correctly answered 22 (100%), as per its remit, regarding the part that was its responsibility and, DANKE correctly answered 20 (~91%) queries, regarding the part that was its responsibility, leading to a final result of 20 (~91%) correctly answered queries.

status/NLIDB	GLAMORISE	%	DANKE	%	Final Result	%
success	22	100%	20	91%	20	91%
failure	0	0%	2	9%	2	9%
Total	**22**	**100%**	**22**	**100%**	**22**	**100%**

Table 6 – DANKE results with ANP dataset

DANKE only presented two errors: in queries **Q12** and **Q13**, instead of identifying oil production, it erroneously identified gas production.

5. Conclusions and Future Work

5.1. Conclusions

The main contribution of this work was the creation of a generic module, called GLAMORISE, to be used in NLIDB systems and which allows the processing of queries with aggregations on the condition that the result of the NLIDB is, or can be transformed into, a result set in the form of a table. This work addressed aggregations with some specificities such as ambiguities, timescale differences, aggregations in multiple attributes, the use of superlative adjectives, basic unit measure recognition, aggregations in attributes with compound names and subqueries with aggregation functions nested up to two levels.

To test the performance of GLAMORISE, a mock NLIDB was implemented to process the set of 22 testing questions using the National Petroleum Agency / Agência Nacional de Petróleo (ANP) dataset. GLAMORISE correctly answered all questions.

The next step was the integration of GLAMORISE with 2 real NLIDBs: NaLIR and DANKE.

With NaLIR, the same set of 22 ANP NLQs were applied. GLAMORISE correctly answered 22 (100%), regarding the part under its responsibility, and NaLIR correctly answered 18 (~82%) queries, regarding the part that was its responsibility, leading to a final result of 18 (~82%) correctly answered queries.

Another set of 17 NLQs testing questions was used based on a Microsoft Academic Search (MAS) dataset. GLAMORISE correctly answered 17 NLQs (100%), regarding the part under its responsibility, and NaLIR correctly answered 11 NLQs (~65%) queries, regarding the part that was its responsibility, leading to a final result of 11 NLQs (~65%) correctly answered.

Finally, with DANKE, the tests were done only with the 22 NLQs of the ANP dataset. GLAMORISE correctly answered 22 (100%), as per its remit, regarding the part that was its responsibility and, DANKE correctly answered 20 (~91%)

queries, regarding the part that was its responsibility, leading to a final result of 20 (~91%) correctly answered queries.

5.2. Publications

The initial research involving GLAMORISE, with a prototype and proof-of-concept with a mock NLIDB, was published in the Proceedings of the XXXV Brazilian Symposium on Databases – SBBD. The paper was awarded second best short paper ("menção honrosa") [Novello and Casanova 2020]. An extended version of this paper was submitted to the Journal of Information and Data Management (JIDM) and is under review at the time of this writing.

One issue that deserves to be commented is that the acronym GLAMORISE appears in the short paper as (GeneraL Aggregation MOdule for RelatIonal databaSEs) and has been modified in this work to (GeneraL Aggregation Module using a RelatIonal database). The rationale behind this modification is that the name was causing confusion and led to the belief that it could only be used with relational databases, while in fact it can be used with relational databases or RDF stores, since the NLIDB that interfaces with the database returns the result set in a tabular format. GLAMORISE, internally, uses a relational database to carry out the rest of the process. This does not prevent its integration with an NLIDB using an RDF store.

5.3. Future Work

The integration with DANKE was in the black box model, that is, without access to its source code. A first suggestion for future work is to carry out a definitive integration with DANKE, that is, integrating the GLAMORISE source code to that of DANKE. For that, we need to port GLAMORISE to Java and natively integrating its ideas and functionalities with DANKE is one line of future work.

To support a more advanced treatment of units of measures, such as conversions, expand the cases involving ellipsis, and other possible cases involving subqueries, in addition to those treated here, is a second fruitful suggestion for future work.

GLAMORISE is prepared, at this time, to accept questions only in English. Another fruitful direction for future work would be to extend GLAMORISE to other languages. This extension should not be difficult since the linguistic patterns used exist in most languages and there are packages of different languages in the NLP libraries used by GLAMORISE and integrated NLIDBs.

Another suggestion for future work path would be the treatment qualitative queries. Aggregation functions can be thought of as being of two types. *Quantitative aggregation functions* have a direct mapping to aggregation functions, such as *max*, *min*, *avg*, *count*, *sum*, and *qualitative aggregation functions*, such as *good*, *bad*, *high*, *low*, etc. , as discussed in [Abhijeet Gupta 2013; Gupta and Sangal 2013]. Databases do not handle qualitative aggregations natively as there is no direct mapping to aggregation functions.

Another type of query that has no direct mapping in the database that would be interesting to be treated in future work is the so-called vague queries [Motro 1988]. A question is *vague* when the result set is empty, but there are close results that could be displayed. Sometimes the result set may not even be empty, but it can be extended to display answers that do not fully contemplate the question, but that serve to complement the answer with similar options.

As a final suggestion for future work is the creation of a complete NLIDB system from scratch based on natural language patterns (the same technique used in GLAMORISE) that can be coupled to GLAMORISE, which would provide the aggregation layer and the new NLIDB the rest of the features.

References

Abhijeet Gupta (2013). Complex Aggregates In Natural Language Interface To Databases.

Affolter, K., Stockinger, K. and Bernstein, A. (2019). A comparative survey of recent natural language interfaces for databases. *VLDB Journal*, v. 28, n. 5, p. 793–819.

Bharati, A., Bhatia, M., Chaitanya, V. and Sangal, R. (2014). Paninian Grammar Framework Applied to English Paninian Grammar Framework Applied to English PG for Indian Languages - A Review. n. May.

García, G. M. (2020). A Keyword-based Query Processing Method for Datasets with Schemas. *PhD Thesis, IT department, PUC-Rio*, n. March.

Gupta, A., Akula, A., Malladi, D., et al. (2012). A novel approach towards building a portable NLIDB system using the computational Paninian grammar framework. *Proceedings - 2012 International Conference on Asian Language Processing, IALP 2012*, p. 93–96.

Gupta, A. and Sangal, R. (2013). A Novel Approach to Aggregation Processing in Natural Language Interfaces to Databases A Novel Approach to Aggregation Processing in Natural Language Interfaces to Databases A Novel Approach to Aggregation Processing in Natural Language Interfaces to Databases. . https://www.researchgate.net/publication/282666480.

Honnibal, M. and Johnson, M. (2015). An improved non-monotonic transition system for dependency parsing. *Conference Proceedings - EMNLP 2015: Conference on Empirical Methods in Natural Language Processing*, n. September, p. 1373–1378.

Honnibal, M. and Montani, I. (2017). spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing.

Izquierdo, Y. T., García, G. M., Menendez, E. S., et al. (2018). QUIOW: A keyword-based query processing tool for RDF datasets and relational databases. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 11030 LNCS, p. 259–269.

Izquierdo, Y. T., García, G. M., Novelli, B. A., et al. (2020). Integrating a geomechanical collaborative research portal with a data & knowledge retrieval platform. *Rio Oil and Gas Expo and Conference*, v. 20, n. 2020, p. 421–422.

Li, F. and Jagadish, H. V. (2014). NaLIR: An interactive natural language interface for querying relational databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. . Association for Computing Machinery.

Li, F. and Jagadish, H. V (2014). Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, v. 8, n. 1, p. 73–84.

Li, F. and Jagadish, H. V (2016). Understanding Natural Language Queries over Relational Databases. *ACM SIGMOD Record*, v. 45, n. 1, p. 6–13.

Motro, A. (1988). Vague: A User Interface to Relational Databases that Permits Vague Queries. *ACM Transactions on Information Systems (TOIS),* v. 6, n. 3, p. 187–214.

Pazos R, R. A., Aguirre L, M. A., González B, J. J., et al. (2016). Comparative study on the customization of natural language interfaces to databases. *SpringerPlus*, v. 5, n. 1.

Pazos R, R. A., Verastegui, A. A., Martínez F, J. A., Carpio, M. and Gaspar H, J. (2018). Translation of natural language queries to SQL that involve aggregate functions, grouping and subqueries for a natural language interface to databases. *Studies in Computational Intelligence*, v. 749, p. 431–448.

Pinheiro, J. P. V, Casanova, M. A. and Menendez, E. S. (2020). Improving the Quality of the User Experience by Query Answer Modification. *Proc. XXXV Brazilian Symposium on Databases - SBBD,*

Pruski, P., Lohar, S., Goss, W., Rasin, A. and Cleland-Huang, J. (1 sep 2015). TiQi: Answering unstructured natural language trace queries. *Requirements Engineering*, v. 20, n. 3, p. 215–232.

Shah, V., Li, S., Kumar, A. and Saul, L. (2019). SpeakQL: towards speech-driven multimodal querying of structured data. p. 1–16.

Tata, S. and Lohman, G. M. (2008). SQAK: Doing more with keywords. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, p. 889–901.

Torres Izquierdo, Y., Monteagudo Garcia, G., Lemos, M., et al. (28 sep 2020). Keyword Search over the COVID-19 Data. In *Anais do XXXV Simpósio Brasileiro de Banco de Dados (SBBD 2020)*. . Sociedade Brasileira de Computação - SBC.

https://sol.sbc.org.br/index.php/sbbd/article/view/13642.

Appendix I – A Sample of the GLAMORISE JSON Pattern Configuration File

Default Pattern

```
"default_pattern": [{"POS": "ADV", "OP": "*"},
                    {"POS": "ADJ", "OP": "*"},
                    {"POS": "NOUN", "LOWER": {"NOT_IN": ["number"]}}
                   ]
```

“More than” - Having Condition Pattern Example

```
‶more than example″:{
  "reserved_words": ["more than"],
  "pre_having_conditions": [">"],
  "specific_pattern": [{"LIKE_NUM": true},
                       {"POS": "ADV","OP": "*"},
                       {"POS": "ADJ","OP": "*"},
                       {"POS": "NOUN","OP": "*"},
                       {"POS": "NOUN"}],
  "pre_cut_text": false
}
```

“by” - Group by Condition Pattern Example

```
"by example": {
  "reserved_words": ["by"],
  "pre_group_by": true,
  "pre_cut_text": false
},
```

“by” followed by an “and” - Group by Condition Pattern Example

```
"by / and example": {
  "reserved_words": ["by"],
  "pre_group_by": true,
  "specific_pattern": [{"POS": "ADV","OP": "*"},
                       {"POS": "ADJ","OP": "*"},
                       {"POS": "NOUN","LOWER": {"NOT_IN": ["number"]}},
                       {"LOWER": "and"},
                       {"POS": "NOUN","LOWER": {"NOT_IN": ["number"]}}],
  "pre_cut_text": false
}
```

“how many” – Aggregation Function Pattern Example

```
"how many example": {
  "reserved_words": ["how many"],
  "pre_aggregation_functions": ["count"],
  "pre_cut_text": false
}
```

“yearly” – Timescale Subquery Pattern Example

```
"yearly example": {
  "reserved_words": ["yearly"],
  "pre_subquery_replace_text": {"yearly": "year"},
  "use_replace_text_as_group_by": true,
  "pre_subquery_aggregation_functions": "sum",
  "pre_cut_text": false
}
```

“most” – Aggregation Function Pattern Example

```
"most example": {
  "reserved_words": ["most"],
  "pre_aggregation_functions": "max",
  "pre_cut_text": true
}
```

“most number of” – Nested Functions Subquery Pattern Example

```
"most number of example": {
  "reserved_words": ["most number of"],
  "use_replace_text_as_group_by": false,
  "remove_external_group_by": true,
  "pre_subquery_aggregation_functions": "count",
  "pre_cut_text": true
}
```