Final Programming Project

A Novel Solution to the Aggregation Problem in a Natural Language Interface to Database

Student: Alexandre F. Novello[1], Advisor: Marco A. Casanova[1], Course Coordinator: Marcos Kalinowski

[1]Department of Informatics, PUC-Rio, Rio de Janeiro, RJ – Brazil

{anovello, casanova, kalinowski}@inf.puc-rio.br

Abstract. *Natural Language Interface to Database (NLIDB) systems usually do not deal with aggregations, which can be of two types: aggregation functions (as count, sum, average, minimum and maximum) and grouping functions (GROUP BY). This work addresses the creation of a generic module to be used in NLIDB systems, which 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. The work covers aggregations with specificities, such as ambiguities, time-scale differences, aggregations in multiple attributes, the use of superlative adjectives, basic unit measure recognition, and aggregations in attributes with compound names.*

Keywords: *Natural Language Interface to Database (NLIDB), Question Answering (QA), Aggregation, SQL.*

1. Introduction

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). 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.

One of the less well-solved tasks in step (4) of this process is the treatment of questions with aggregation involving grouping (GROUP BY clause) and aggregation functions such as count, sum, average, minimum and maximum, especially when the question is on another time-scale in relation to the stored data and it is necessary to perform a conversion. For example, consider the question: *"What was the average annual compensation for employees in 2019?"* 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 2019 tuples of compensation per employee and only then perform the average.

NLIDB systems usually do not deal with the treatment of aggregations, but they produce good results for normal queries. The contribution of this work is the creation of a generic module to be used in NLIDB systems, called GLAMORISE (GeneraL Aggregation MOdule for RelatIonal databaSEs). 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 even 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 the JSON format. We address some aggregations with specificities including ambiguities, time-scale differences, aggregations in multiple attributes and aggregations in attributes with compound names. To test our module integrated with an NLIDB, we use a mock NLIDB that will be described later.

The rest of this work is organized as follows. Section 2 is a literature review. Section 3 describes our solution. Section 4 presents the schedule of the project. Section 5 addresses the used technology. Section 6 describes the documentation, Section 7 evaluates the performance and presents examples of two types of aggregations, among others, that were tackled in this work. Section 8 presents the tests and the benchmark questions. Section 9 describes the installation process. Section 10 contains the link for Github and Google Colab for reproducibility purposes. Finally, Section 11 contains the conclusions and suggestions for future work.

2. Related Work

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 deal with similar problems, aiming at the use of keywords for the final translation to SQL. The difference is that their work, as well as others that we will refer to this section, are complete and monolithic NLIDBs, while ours aims at enabling existing NLIDBs to handle aggregations, which they do not perform well with this type of question.

NaLIR [Li and Jagadish, H. V. 2014; Li and Jagadish 2016; Li and Jagadish, H. V 2014] is a generic NLIDB which is capable of handling aggregations, nesting and various types of joins is presented. The Stanford NLP Parser is used to convert from natural language into a parser tree. The approach followed is to take feedback from the user, returning 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. In our work, we prefer the approach of generating a structured query automatically from the natural language query.

In [Gupta et al. 2012] a novel approach is 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 received an English version. They argue 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 is 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.

The work we are going to discuss now was also developed in two stages, similarly to the previous work. 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 the 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 for this, 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 we also identified.

3. GLAMORISE – A Proposed Solution

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 architecture of our system is shown below in Figure 1.

Initially, in the user interface, the user types a query *Q* in natural language. GLAMORISE **Preprocessor** removes the aggregation elements and transform *Q* into a normal query *Q'* and registers all the elements related to the aggregation in its metadata database, to be subsequently used by the **Aggregation Processor**. Then, *Q'* is sent to the conventional NLIDB, which returns a result set *R* without any aggregations. The metadata of the data result set can also be retrieved to perform some more intricate questions. After that, GLAMORISE **Aggregation Processor** processes the aggregation over the result set *R*, creates the final result set with aggregations *R'*, and returns it to the user interface.

The purpose of the **Preprocessor** is to map the keywords in natural language to the respective aggregate functions. The other main mission is to identify whether the query in natural language also has grouping (GROUP by clause) conditions. These keywords are removed or substituted from the query to guarantee that the conventional NLIDB will not be confused by their existence, leading to incorrect mappings.

The Aggregation Processor is responsible for building the query itself in SQL, being responsible 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 (GROUP BY clause), reading the metadata to determine if there is grouping, which fields are involved, mapping them in the result set, and also if there is a HAVING clause and its conditions. Another step is conducted to analyze any timescales that should be converted with the use of an aggregate function (sum) and grouping.

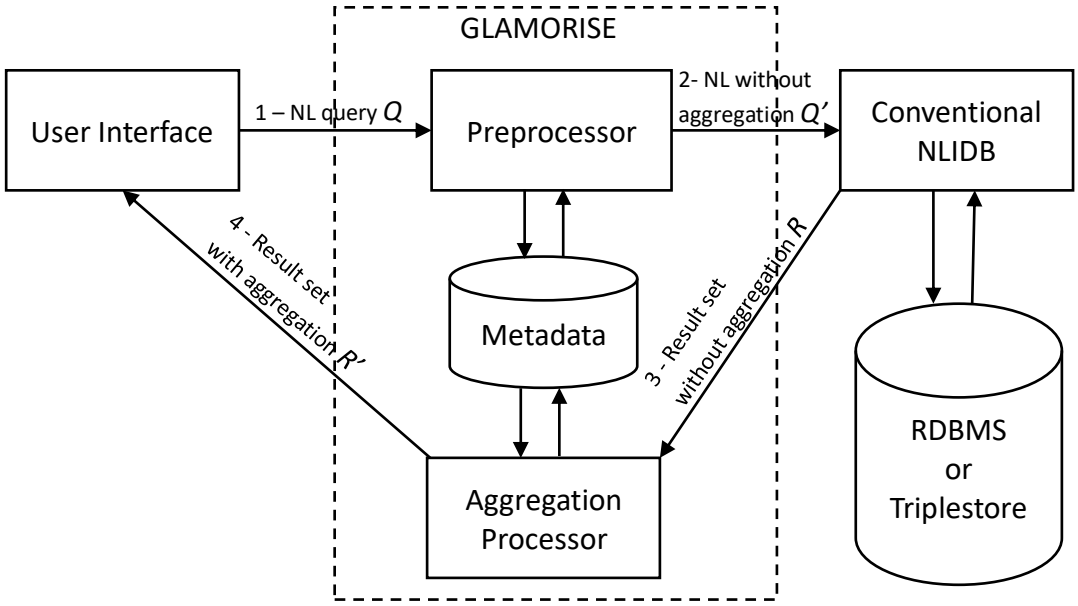


Figure 1. GLAMORISE Architecture

There are many textual patterns in natural language queries that can be translated as aggregations. In this work, we focused on treating aggregations with some types of ambiguities, time-scale differences, aggregations in multiple attributes, aggregations with the use of superlative adjectives, basic unit measure recognition and aggregations in attributes with compound names. It is beyond the scope of this work to deal with elliptical aggregations, and queries with subqueries using aggregations (other than the time-scale problem stated and resolved). It was decided not to deal with qualitative queries. Aggregate functions can be thought of as being of two types. *Quantitative aggregation functions* have a direct mapping to the aggregate functions, such as *max*, *min*, *avg*, *count*, *sum*, and *qualitative aggregation functions*, such as *good*, *bad*, *high*, *low*, etc. Databases do not handle qualitative aggregations natively as there is no direct mapping to aggregate functions. The first time we saw the use of this type of function was in [Abhijeet Gupta 2013; Gupta and Sangal 2013]. The problem we identified in dealing with this type of

query is the lack of standardization. What is good or near for one person is not good or near for another.

4. Schedule

In the following table the project schedule is presented with each activity and its respective start and end dates:

Table 1 Schedule

Activity	Start Date	End Date
Read NLIDB papers	Mar 26	Apr 25
Project	Apr 26	May 19
Codification	May 01	Jul 02
Tests	Jul 03	Jul 08
Refactoring	Jul 04	Jul 08
Documentation	Apr 26	Jul 08

5. Technology

GLAMORISE was implemented in Python[1] with the help of spaCy to handle natural language processing. We also used regular expressions (regex) to identify some patterns that did not depend on the parse tree.

spaCy[2][Honnibal and Montani 2017] is an open-source software library for advanced natural language processing, written in Python and Cython[3]. 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 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[4] is an open source library which provides easy-to-use high-performance data structures and data analysis tools for the program language.

Both the Mock NLIDB and GLAMORISE use SQLite as their database. SQLite[5] is a relational database management system (RDBMS) written in C. In contrast to 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 PostgreSQL syntax.

The system was developed using PyCharm[6], which is an integrated development environment (IDE) used in computer programming, specifically for the Python language.

1 https://www.python.org/

2 https://spacy.io/

3 https://cython.org/

4 https://pandas.pydata.org/

5 https://www.sqlite.org/

6 https://www.jetbrains.com/pycharm/

It was developed by the Czech company, JetBrains. A Jupyter Notebook was also produced to simplify execution and reproducibility, and is hosted at Google Colab. Jupyter Notebook is a web-based interactive computational environment for creating Jupyter notebook documents. With its technology it is possible for the developer to program directly in a web environment. Google Colab is a service to store Jupyter Notebooks as SaaS, giving access to dedicated cloud machines.

Git and Github were used as version controller and version control repository, respectively.

The test library used was Pytest.

6. Documentation

6.1. Use Case Diagram

In Figure 2 the use case of the system is presented. The system has just one use case because it has a very specific purpose. The user asks the system a natural language question and the system returns a tabular result set to the user.

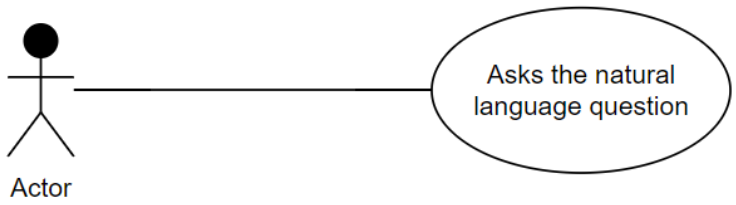


Figure 2. Use Case Diagram

6.2. Class Diagram

In Figure 3 the class diagram is presented. The classes presented by the system are:

- **GLAMORISE –** this is the main class of the system. Its main purpose is to receive the natural language query from the user, identify the aggregation elements, and prepare the new natural language query without the aggregation elements to pass to the conventional NLIDB. After receiving the result set without aggregation from the NLIDB, it persists the result set in a local database (SQLite) and generates the SQL with the aggregation elements producing a new result set with aggregation. This new result set is presented to the user using the Pandas library.
- **GLAMORISEMockNLIDB** – this is the child class of GLAMORISE. This is the class that is instantiated because the GLAMORISE class has some abstract methods that are defined in GLAMORISEMockNLIDB. This class is aware of the implementation of the NLIDB and is responsible for undertaking the integration between them.
- **MockNLIDB –** to test the system without the need of a real NLIDB, a Mock NLIDN was built. This Mock NLIDB is prepared to receive a set of natural language queries and return the appropriate SQL, without aggregation, to GLAMORISE.

- **SimpleSQLLite** – a simple class built merely to facilitate the interaction with the SQLite databases. GLAMORISE has one database and MockNLIDB has another.

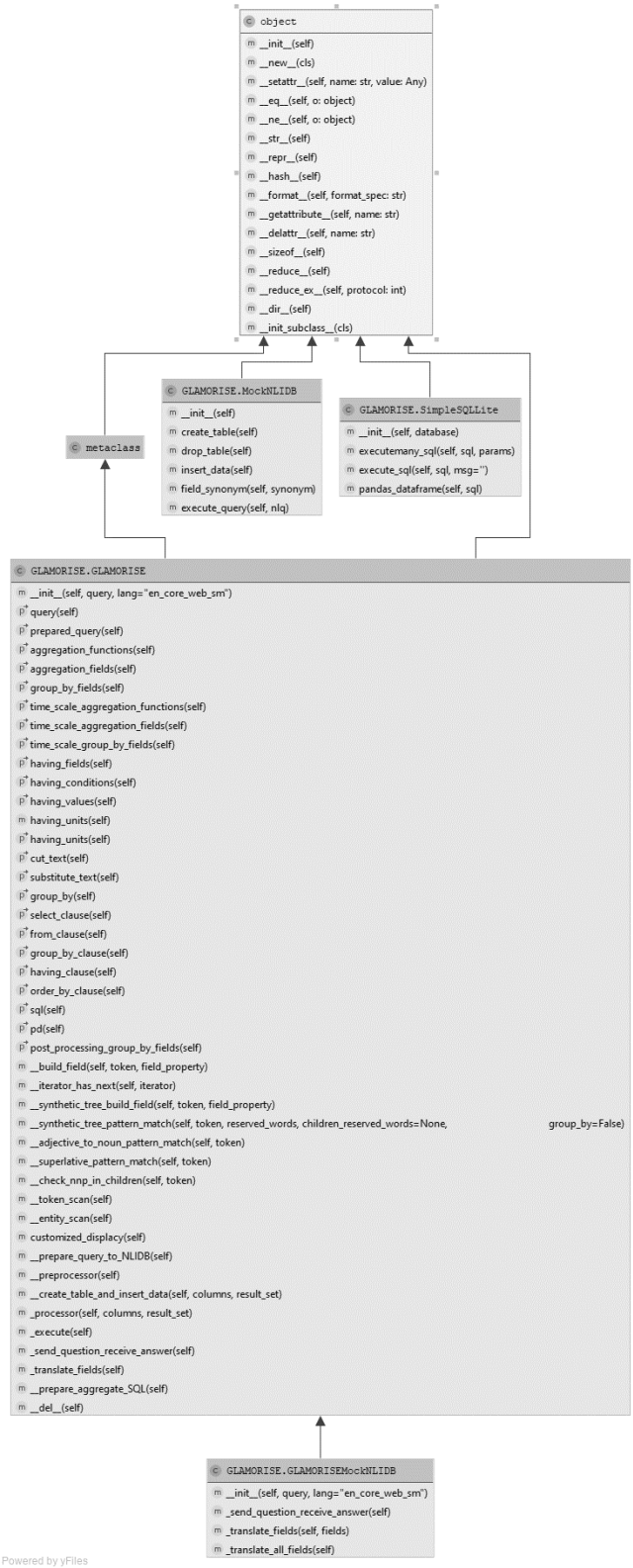


Figure 3. Class Diagram

6.3. Sequence Diagram

Figure 4 presents the main activity of the system. It is a more formalized version, following the UML conventions, of Figure 1.

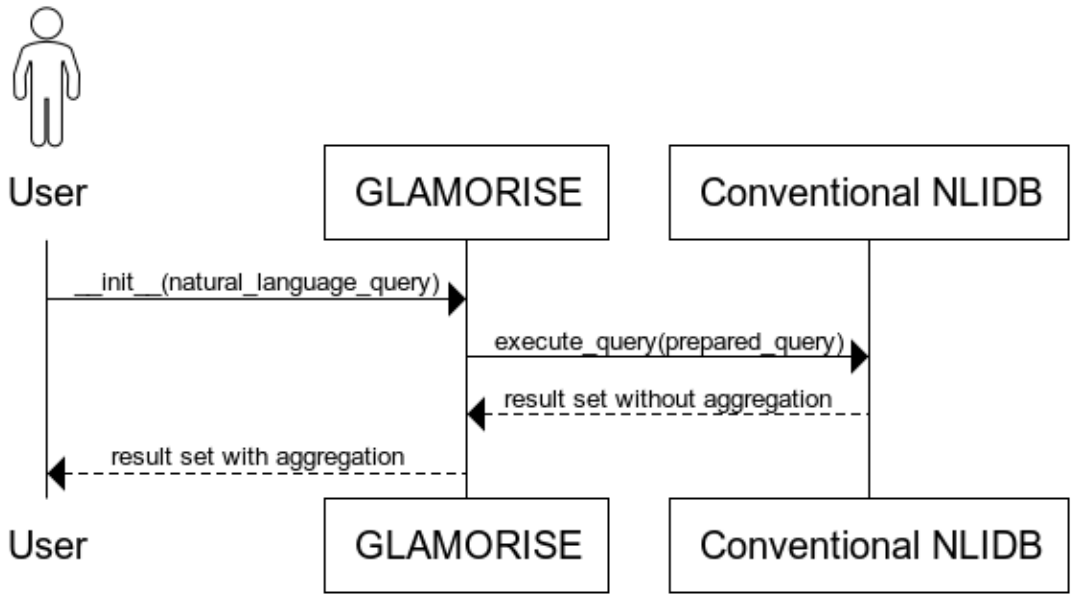


Figure 4. Sequence Diagram

6.4. Database

Mock NLIDB

To test our module, we built a mock NLIDB. This Mock NLIDB has a SQLite database associated with it (NLIDB.db). We assumed it returns a result set in the form of a table, that is, a single non-normalized table. To simplify our discussion, we assumed that the table returned, in all NLIDB examples, can have, at most, the attributes shown in Table 2. This is the source table where it builds the result set. Depending on the question asked, the result set may not return all these attributes but a subset of them. The knowledge domain of this table is the production of oil fields in Brazil. The data is released by the National Petroleum Agency (ANP)[7].

Table 2. ANP Table

Name	Type
FIELD	TEXT
BASIN	TEXT
STATE	TEXT
OPERATOR	TEXT
CONTRACT_NUMBER	TEXT

7 http://www.anp.gov.br/

OIL_PRODUCTION	REAL
GAS_PRODUCTION	REAL
MONTH	INTEGER
YEAR	INTEGER

We have two metadata tables. Table 3 provides the common reserved words that should be associated with the fields in the ANP table.

Table 3. NLIDB_FIELD_SYNONYMS Table

Name	Type
SYNONYM	TEXT
FIELD	TEXT

Table 4 is responsible for providing the mock functionality of the Mock NLIDB. It associates a natural language query with its respective SQL, simulating the operation on a real NLIDB.

Table 4. NLIDB_SQL_FROM_NLQ Table

Name	Type
NLQ	TEXT
SQL	TEXT

GLAMORISE

The SQLite database associated with the GLAMORISE class (GLAMORISE.db) starts empty, and just one table is created during its execution. The name of this table is NLIDB_RESULT_SET and is the result set returned as JSON by the NLIDB persisted in a relational table. The structure of the table is a subset of the Table 2. ANP Table.

7. Performance

To test the performance of GLAMORISE, we implemented a mock NLIDB prepared to receive the set of testing questions with completely different etymologies. First, we confirmed that the questions were preprocessed correctly, removing or substituting words (aggregation elements) as necessary. Second, we confirmed that the generated SQL queries with aggregations were correct.

Taking into account the 22 questions that were asked, all of them were answered correctly.

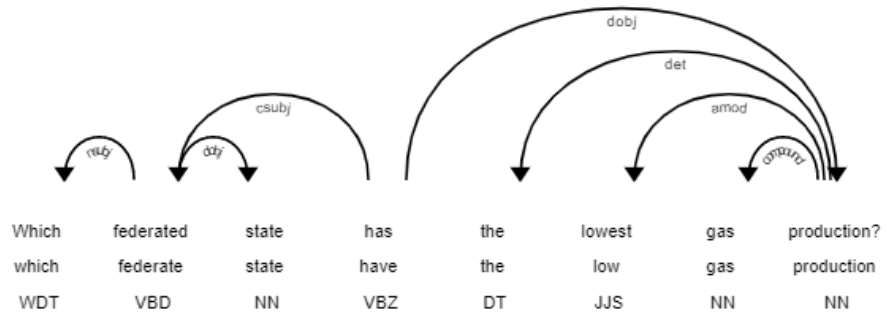
Two examples of different sentences types are presented below to show how GLAMORISE works. Currently, GLAMORISE is prepared to tackle natural language queries with aggregations specificities such as ambiguities, time-scale differences, aggregations in multiple attributes, the use of superlative adjectives, basic unit measure recognition and aggregations in attributes with compound names.

Superlative Adjectives

The superlative adjectives are suppressed and, depending on the type of superlative, a *min* or *max* function is added to the aggregate functions metadata, and 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 Parse Tree



GLAMORISE Internal Properties

```
aggregation_fields = ['gas_production']
aggregation_functions = ['min']
cut_text = ['lowest']
post_processing_group_by_fields = ['state']
prepared_query = 'Which federated state has the gas production?'
sql = 'SELECT state, min(gas_production) as min_gas_production FROM NLIDB_result_set GROUP BY state ORDER BY state'
```

Figure 5. Superlative adjective example

The *min* aggregate function is identified because of the presence of the superlative adjective "lowest"; also, "gas production" is identified as the aggregate field that is related. The relation is obtained by the parse tree.

Aggregations with time-scale differences

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. Looking at the Table 2 again, two attributes are noted, one for the year and another for the month, in addition to production (oil or gas). That is, each tuple associates production with one year and one month. The SQL of this query would be:

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

The most attentive readers will notice that this query is different from the previous examples. Namely, there are two aggregation functions. The first performs the sum of the grouped attribute, “year”, and then the average of all years is calculated.

For convenience, in this first implementation, we are using the SQLite to store the metadata and process the aggregation in the result set. At the moment, the SQLite does not support nested aggregate functions like “AVG(SUM(*field*))”, so our query has to be translated to:

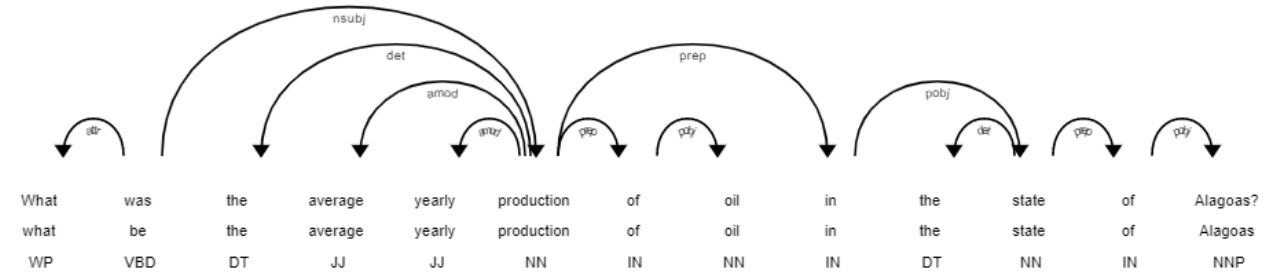
```
SELECT AVG(sum_oil_production) as avg_sum_oil_production
FROM  (SELECT SUM(oil_production) as sum_oil_production
FROM nlidb_result_set
WHERE state = 'Alagoas'
GROUP BY year)
```

The **Preprocessor** converts the adjective, in the case of the example "yearly", to its corresponding noun, in ’this case "year". When we receive the NLIDB result set for this type of question, we also receive a metadata result set with information regarding the timescale in which the data is stored (daily, monthly, yearly, etc.). If the question was asked in a different scale, the **Processor** does the aggregation accordingly (SUM(*field*) and GROUP BY); if it were done on the same scale, there is no second aggregate function or grouping.

In the figure below we can see how the **Preprocessor** recognizes the sentences and separates the "average" interpretation, which is the normal aggregation that will be made, from the "yearly" interpretation, which is the time-scale aggregation that will be made, depending on the time-scale that is stored in the database.

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

spaCy Parse Tree



GLAMORISE Internal Properties

```
aggregation_fields = ['oil_production']
aggregation_functions = ['avg']
cut_text = ['average']
prepared_query = 'What was the year production of oil in the state of Alagoas?'
sql = 'SELECT avg(oil_production) as avg_oil_production FROM (SELECT sum(oil_production) as oil_production FROM NLIDB_result_set GROUP BY year)'
time_scale_aggregation_fields = ['oil_production']
time_scale_aggregation_functions = ['sum']
time_scale_group_by_fields = ['year']
```

Figure 6. Aggregations with time-scale differences example

8. Tests

We implemented one functional test that has two asserts. One assert tests if the natural language query parsed by GLAMORISE to the NLIDB removed the aggregation elements correctly. The second assert tests if the SQL generated by GLAMORISE was correct, with all aggregation elements correctly translated to its SQL clauses. We repeated his test for each of the 22 queries. Table 5 below presents all the natural language queries that were tested.

Table 5. Natural Language Queries (NLQ)

NLQ
What was the production of oil in the state of Rio de Janeiro?
What was the average monthly production of oil in the state of Rio de Janeiro?
What was the average yearly production of oil in the state of Alagoas?
How many fields are there in Paraná?
What was the maximum production of oil in the state of Ceará per field?
What was the minimum gas production in the state of São Paulo per basin?
What was the average monthly oil production by the operator Petrobrás?
What was the mean yearly gas production per field?
What was the mean gas production per month per field?
What was the per month mean gas production per field?
What was the per field mean gas production per month?
What was the mean monthly petroleum production by field in the state of Rio de Janeiro?
What was the mean yearly petroleum production by field by Rio de Janeiro?
What was the average monthly production of oil per field in the state of Rio de Janeiro and year 2015?
What was the average yearly production of oil per field and state in the year in 2015?
What was the mean gas production per field with production greater than 100 cubic meters?
What was the mean gas production per basin with production less than 1000 cubic meters?
Which field produces the most oil per month?
Which basin has the highest yearly oil production?
Which federated state has the lowest gas production?
Which state of the federation has the lowest gas production?

The code of the text is below in Figure 7:

```
class TestGLAMORISE(TestCase):
    def test_GLAMORISE_methods(self):
        # set of questions to test
        with open('./datasets/pfp.csv', encoding="utf-8") as csv_file:
            csv_reader = csv.reader(csv_file, delimiter=';', quotechar="'")
            #jump the title line
            next(csv_reader)
            for row in csv_reader:
                #the NLQ is the first column of the CSV
                nl_query = row[0]
                glamorise = GLAMORISEMockNLIDB(nl_query)
                try:
                    # assert the NLQ generated by GLAMORISE to the NLIDB is equal to the expected value
                    # (second column of the CSV)
                    assert glamorise.prepared_query.lower() == row[1].lower()
                finally:
                    # print anyway, just for convenience (pytest cuts the string)
                    print('\nPrepared NLQ to NLIDB\nExpected: ', row[1].lower())
                    print('Actual:   ', glamorise.prepared_query.lower())
                try:
                    # assert the SQL generated by GLAMORISE is equal to the expected value
                    # (forth column of the CSV)
                    assert glamorise.sql.lower() == row[3].lower()
                finally:
                    # print anyway, just for convenience (pytest cuts the string)
                    print('\nGLAMORISE SQL\nExpected: ', row[3].lower())
                    print('Actual:   ', glamorise.sql.lower())
            csv_file.seek(1)
            print('{} NLQ questions tested'.format(sum(1 for line in csv_reader)))
```

Figure 7. Test Code

In Figure 8 is possible to check if the test was executed successfully.

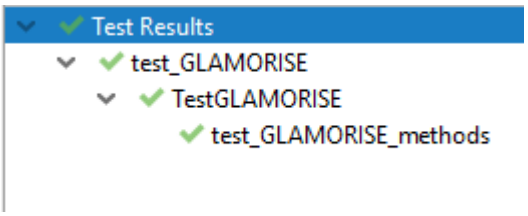


Figure 8. Test Execution

9. Installation

The system is still a prototype and is going to be further developed until the end of the students' master's dissertation. For now, the only way for it to be integrated with a conventional NLIDB is by reusing its code and changing the class code: MockNLIDB and GLAMORISEMockNLIDB.

10. Reproducibility

All code developed in Python, as well as the questions used as a benchmark can be found in the project's repository on Github at: https://github.com/novello/final_programming_project

The Google Colab version of the project can be found at: https://colab.research.google.com/drive/1ayyI7tgvqIiIrm24_AZ-GDD9t8I6H7M6?usp=sharing

11. Conclusions and Future Work

In this work, a novel solution to the aggregation problem in a natural language interface to database is presented. The contribution of this work is the creation of a generic module, called GLAMORISE, to be used in NLIDB systems. This module allows the processing of queries with aggregations, on condition that the result of the NLIDB is, or can be transformed into, a result set in the form of a table. We addressed some aggregations with specificities such as ambiguities, time-scale differences, aggregations in multiple attributes, the use of superlative adjectives, basic unit measure recognition and aggregations in attributes with compound names.

As future directions, we contemplate, as a first step, dealing with more complex unit measure recognition, followed by tackling subqueries and elliptical aggregations. Ellipsis is the suppression of a term that can be easily understood by the linguistic context or situation.We also thought about using questions from previous QALD challenges as a benchmark.

References

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

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.

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.

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.

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.

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.