FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Using Neo4j DB system to store and query linguistic pattern
Student:	Vigneshwar Manoharan
Supervisor:	prof. Dr. Ing. Petr Kroha, CSc.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2021/2022

Instructions

The methods of Text Mining elicitate information from textual formulated requirements specification in our case. It uses linguistic patterns for this purpose. These patterns can be represented as oriented graphs where edges are tokens (words and interpunction) and vertices represent dependencies in sentence clauses.

Outline:
1. Introduction to Neo4j DB system.
2. Introduction to linguistic patterns used.
3. Design and implementation of Neo4j DB to store patterns.
4. Design and implementation of a custom API interface and a web interface for linguistic patterns storing and querying.
5. Tests and experiments.
6. Results evaluation and conclusion.

Electronically approved by Ing. Michal Valenta, Ph.D. on 3 November 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Using Neo4j DB system to store and query linguistic pattern

VIGNESHWAR MANOHARAN B.E.

Department of Software Engineering
Supervisor: Prof. Dr. Ing. Petr Kroha, CSc.

December 27, 2021

Acknowledgements

This thesis work would not have been possible without assistance from Mr. Kroha and Mr. David Senkyr, and I'm also grateful to my family and friends who assisted me during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on December 27, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

MANOHARAN, VIGNESHWAR. *Using Neo4j DB system to store and query linguistic pattern.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

V této práci prezentuji svou implementaci ukládání lingvistických vzorů jako orientovaného grafu do databáze Neo4j a dotazování se na odpovídající vzory. Dále to bude využito v jedné z činností dolování textu, které gramaticky kontrolují nestrukturovaný text, a to primárně se značkováním slovních druhů a analýzou závislostí mezi každým slovem věty za účelem odhalování nepřesností, které se vyskytují v textu a které jsou způsobeny nejednoznačností, neúplnost a nedůslednost. Tento proces používá metodu rozpoznávání založenou na vzorech k identifikaci vzorů v textu a poté jej porovnává s definovanými vzory, aby se zjistily nepřesnosti. Protože tyto textové vzory věty jsou reprezentovány jako orientovaný graf, budou uloženy v databázi Neo4j, která obsahuje slova, slovní druhy a interpunkci jako uzly. Závislosti mezi každým uzlem budou uloženy jako vztahy a poté bude provedeno porovnávání dotazu (vzoru vět) s předdefinovaným uloženým vzorem. Toto slouží ke kontrole, které předdefinované vzory jsou podgrafy dotazu (vzor vět). Takže tyto výsledky budou použity v další fázi procesu dolování textu k detekci a opravě nepřesností, které se vyskytují v textu.

Klíčová slova Databázový systém Neo4j, Lingvistické vzory, TEMOS, Py2Neo, Cypher query

Abstract

In this thesis, I present my implementation of storing linguistic patterns as an oriented graph in a Neo4j database and querying it to get matched patterns. Furthermore, this will be used in one of the text mining activities that grammatically inspect the unstructured text, primarily with the part of speech tagging and dependency parsing between each word of a sentence to detect inaccuracies that occur in a text that are caused by ambiguity, incompleteness, and inconsistency. This process uses a pattern-based recognition method to identify the patterns in a text and then matches it with the defined patterns to detect inaccuracies. Since these textual patterns of a sentence are represented as an oriented graph, they will be stored in the Neo4j database which holds words, parts of speech, and punctuation as nodes. Dependencies between each node will be stored as relationships, and then the matching of Query (sentence pattern) with a predefined stored pattern will be done. This is to check which predefined patterns are subgraphs of the Query (sentence pattern). So, these results will be used in a further stage of the text mining process to detect and fix the inaccuracies that occur in a text.

Keywords Neo4j database system, Linguistic patterns, TEMOS, Py2Neo, Cypher query

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The process of gathering requirements from stakeholders in order to construct software is well-known as *Requirement Specification*. Additionally, while some general methods such as mathematical specifications and graphical notations are used to document these requirements, but an easy and informal way to write requirements in a simple manner is through text or natural language, as it enables both stakeholders and analysts to understand clearly. [1] However, textual requirement specifications have some problems, including ambiguity (words may have various meanings), inconsistency (text may contain inconsistencies), and incompleteness (lack of specific parts of information). As a result, textual requirements necessitate both software engineers and computational linguistics experts who can examine and analyze the language using their semantic understanding. [2] Similarly, to detect problems in a textual requirements specification, a Natural Language Processing (NLP)-powered tool called TEMOS (Textual Modelling System) is designed and implemented in a research paper [2]. They extracted grammatical methods for recognizing patterns in a text using that tool, which includes information on how words are related via dependencies, the type of part of speech to which the words belong, and so on. Following that, an investigation was undertaken utilizing certain predefined linguistic patterns in order to resolve the previously described concerns with textual requirements. Additionally, this tool is capable of automatically generating fragments of the UML class model from the textual requirements specification. Furthermore, to detect ambiguities in a textual requirement, ambiguity patterns have been constructed. [3] Thus, if the grammatically checked textual requirements fit an ambiguity pattern, TEMOS will issue a warning message to the user and may even provide suitable remedies in some circumstances. Similarly, incompleteness patterns are built to check for missing information in textual requirements, which results in an incompleteness problem. [4] and patterns are also constructed to identify

inconsistency produced by contradiction, which occurs when a phrase contains one assertion but also contains its negative. [5] Certainly, incomplete information is also a cause of inconsistency.

Thus, if specified patterns and textual requirements match, a TEMOS tool will create a warning message informing users or domain experts of the inaccuracies and the need for corrections.

How, though, will this operation be connected to this thesis? Indeed, because linguistic patterns are represented as an oriented graph, they chose to store them in a Neo4j graph database for pattern matching. Additionally, the Neo4j database will be integrated with the TEMOS tool, which will evaluate textual requirements by converting them to graphs and querying in Neo4j. [6] Thus, my role is to store both the Query (sentence) patterns to be analyzed and the predefined patterns in a Neo4j database through interaction with the TEMOS tool. Then, using Neo4j's Cypher query, pattern matching will be performed and the necessary results obtained for additional textual mining operations.

1.1.1 Why Graph Database and especially Neo4j is perfect to store linguistic patterns?

The defined linguistic patterns must be stored in a database and compared against the structured format of a textual requirement determined by grammatical inspection. In general, these patterns are represented as oriented graphs, with the *vertices* representing the *words* and the *edges* representing the *relationships* between them. As a result, it is straightforward to store in the Neo4j graph database. Everything is saved and shown in Neo4j as graphs with *nodes* and *relationships* between them. The textual requirements or sentences to be validated will be loaded into Neo4j as nodes and relationships. It stores *parts of speech tagging* as *properties* of nodes, and each word gets related to other words depending on their *dependencies.*

According to the paper, [6] an analysis was conducted to identify semantically similar sentences, i.e., a sentence from a textual requirements specification has a similarity to a sentence from an external source, and the similar sentence can semantically enrich a sentence. Using RDF, they clustered sentences from textual requirements and external sources into triples and created subgraph relations between them to determine similarity. Then a semantic enrichment method is used to speed up the search and generate questions for domain experts to avoid incompleteness in the textual requirement specification.

Due to the fact that the patterns are represented as oriented graphs, the Neo4j database was chosen to store the sentence to be analyzed, and these sentences will be saved in a database as a graph according to their lexical and structural content. The transitive closure may thus be performed directly in

Neo4j, and subgraph matching through cypher queries is also allowed. Following that, these matched results would be utilized to do semantic enrichment.

1.2 Objectives

This thesis's primary purpose is to develop a Neo4j database for storing all defined linguistic patterns as graphs with vertices and edges. In Neo4j, vertices are referred to as nodes, while edges are used to describe the relationships between nodes. Then, as explained in section 1.1.1, the analyzed sentence will be generally inserted into the database using a cypher query. The sentence will be transmitted by an application called TEMOS, which uses natural language processing to perform grammatical inspection on a sentence based on its parts of speech and dependencies. Due to the fact that TEMOS is developed in Python, it delivers a sentence as a list including attributes such as text, parts of speech tagging, dependencies, and the text's head.

My goal is to write Python code that iterates over the sentence list, and then, using a Python-Neo4j interface, to store the iterated sentences in Neo4j as nodes and the lexical content of a text as node characteristics such as text, pos, dep, head, and so on. Then, depending on the property called dependencies, which is identified as a relationship type, relationships will be constructed between nodes, and the head of a text should specify which word will be the sender node. Additionally, all node and relationship creation is dynamic, which means that the Cypher query takes any sentence and automatically stores it in a database as nodes and relationships. Then, using a cypher query, I'll do subgraph matching on a sentence graph and report the results containing the predefined pattern name that matches the sentence.

These matched patterns will be utilized in association with the further text mining activities described in section 1.1 and section 1.1.1 precisely to facilitate the semantic enrichment process and to identify flaws in textual requirement statements.

1.3 Structure of Thesis

This thesis will be classified into various parts. It begins with an introduction in which I discuss the topic's background and motivations. Then, the rationale for using the Neo4j database to store linguistic patterns is discussed in section 1.1.1, and section 1.2 includes the intended goals to be achieved.

Next to the introduction, the following chapter 2 will be the state of the art chapter that includes a brief description of the required software and information pertinent to this thesis, such as Natural Language Processing (NLP), the Neo4j database, and linguistic patterns.

chapter 3 describes the analysis that was undertaken to collect necessary resources and illustrates how the database will be constructed to precisely

store and query linguistic patterns. Additionally, I will describe the Neo4j-Python interface in this chapter.

Following that, the chapter 4 describes the deployment of the required software and tools to begin implementation.

In chapter 5, I will detail my implementation of storing *predefined linguistic patterns* in a database using the Py2neo interface and loading a *Query(sentence)* sent from a TEMOS tool along with its lexical content based on grammatical inspection as a graph entry in the Neo4j database. Importantly, a dynamic pattern matching between the *Query (sentence)* graph and *predefined linguistic patterns* performed by a Cypher query will be demonstrated fully. Additionally, test cases will be built to verify that Cypher query operate properly. Following that, any remaining issues will be resolved by amending the Cypher query. Once the dynamically performed Cypher query is complete, I will finish the implementation by integrating it into the Python code that does pattern matching and returns results.

The completely developed Python code will be tested at chapter 6. It begins by outlining the requirements that must be met. The program will then be checked for functionality and accuracy using the pytest framework.

Finally, I conclude my argument with a chapter section 7.1. This chapter will summarize my thesis implementation, confirming that I accomplished all of the thesis's prerequisites. Furthermore, I will discuss probable future developments.

Chapter 2

State-of-the-art

It is necessary to provide a detailed explanation of the linguistic patterns that I am going to store in a database and query using it, as well as a brief description of the textual requirements and issues that arise in them, as well as other tools that are required, most notably the Neo4j database, which serves as the foundation for this thesis implementation. Thus, all of these points will be discussed in detail in this chapter.

2.1 Natural Language Processing

Natural Language Processing (NLP) is a subfield of Artificial Intelligence concerned with machine learning to comprehend natural language text and spoken words in the same manner that humans do. [7] It is a synthesis of computational linguistics with *statistical, machine learning, and deep learning models of human language*. Through the use of NLP, computer programs may be programmed to translate across languages and to respond to spoken commands. Natural language processing converts unstructured text or speech data to a structured data format. This is accomplished by the identification of named entities and also through the identification of word patterns through the use of procedures such as *tokenization, stemming*, and *lemmatization*, which investigate the roots of words. [8]

2.1.1 NLP Tasks

NLP performs a variety of activities that break down a text and enable applications to comprehend and evaluate it, as well as deal with its grammatical structure, meaning, and usage. Several of these tasks include the following:

- **Speech recognition** software that translates spoken words into text. It's advantageous for applications that respond to voice instructions.

- **Part of speech tagging** is subject to grammatical examination. It recognizes and annotates the parts of speech contained in a word or text, such as a *noun, verb, adverb*, and so on.

- **Word sense disambiguation**, which identifies words that may have different meanings and thus leads to *ambiguity*. It determines the unclear word using a semantic analysis procedure.

- The phrase **Co-reference resolution** refers to the process of determining if two terms relate to the same thing.

- **Sentiment analysis** is used to examine texts that contain subjective traits, feelings, and attitudes.

- The term **Named entity recognition** is sometimes used to describe to the job of extracting and classifying important information (entities) from a text. It might be a single word or a collection of words. [9]

  For instance, it recognizes the name *John* as a person and the firm *Oracle* as a company.

2.1.2 Levels of NLP

Natural language processing is separated into several phases or tiers, which are as follows:

- The study of speech sounds within and between words is referred to as **Phonology.** [10] This level, however, is outside the scope of this thesis since I will be primarily utilizing structured textual patterns based on NLP.

- At the **morphological level**, the *nature of words* has been investigated, including *word structures* and *word creation*. Words are comprised of morphemes; they are the smallest units of meaning. For example, the word *unhappiness* may be deconstructed into morphemes such as prefix-un, suffix-ness, and stem-happy, which is also known as a *free morpheme*. The *bound morpheme* (prefix and suffix) necessitates the attachment of this free morpheme. That is, the prefix and suffix are determined by the root word. [11]

- **Lexical** level enables NLP to analyze words based on their lexical meaning (individual words) and parts of speech such as nouns, adverbs, and verbs, among others. It makes use of a language called *lexicon*, which is a collection of individual *lexemes*, which might be a single word or a combination of words.

- The **Syntactic level** of analysis examines the words in a sentence to determine its grammatical structure. It necessitates the use of both a *grammar* and a *parser*. This level's output exposes both structural links between words and their parts of speech tagging (POS). [10] It makes use of parsing, which is the process of dividing a phrase into constituents and describing their syntactic responsibilities. After parsing a sentence, it generates a parse tree that comprises *part of speech tagging*, phrases (with the subject as a noun phrase and the predicate as a verb phrase), and dependency links between words.

  Both the *lexical* and *syntactic* analyses are more pertinent to this thesis. Because I will be storing textual patterns that were generated using a tool called *spacy NLP* based on *lexical* and *syntactic* structure. I shall discuss this method in further detail in my subsequent explanations.

- The **Semantic level** assists in determining the precise meaning of a sentence by establishing a relationship between the syntactic aspects discussed previously, and also involves *disambiguation of words*, which have many definitions or meanings. This level focuses on the proper interpretation of sentences rather than on the analysis of individual words or phrases. [11]

- The other levels, which are unrelated to this thesis, include **Discourse**, which analyzes the structure and meaning of a text beyond a single sentence and makes connections between words and sentences, and **Pragmatic**, which takes real-world knowledge into account and understands how it affects the meaning of words.

2.1.3 Tools and approaches

Natural language processing includes a variety of tools that help us grasp a language and demonstrate how it works in certain scenarios. [12] There are several NLP tools available on the market. I'll describe various Python-based tools and their relationships to the backdrop of linguistic patterns, which I'll use for a Neo4j database in this thesis.

1. **Python with Natural Language Toolkit**: The Python programming language has a variety of tools and libraries for doing natural language processing tasks. The majority of them are included in the Natural Language Toolkit, which is open source software that includes libraries and tools for constructing natural language processing algorithms. [7] NLTK contains libraries for natural language processing tasks such as

   - **Sentence parsing** (as mentioned before in the syntactic level part of section 2.1.2)
   - **Word segmentation**

- **Lemmatization and stemming** (the process of reducing words to their root words)
- **Tokenization** for breaking phrases, sentences, and paragraphs into tokens to help a machine better understand a text.
- **Semantic reasoning** is used to reach logical conclusions based on facts taken from a text.

2. **Stanford Core-NLP**: It is a general-purpose text analysis tool, similar to NLTK. The primary benefit of Stanford Natural Language Processing is its tremendous scalability. Because of high scalability it is better for,

   - Extracting data from open source websites (social media, user-generated reviews)
   - Sentiment analysis
   - Conversational interfaces such as chatbots
   - Text processing and production, which are beneficial for customer support, e-commerce, and so forth.

3. **Spacy-NLP**: Spacy is the NLTK tool's next stage. While NLTK is rather sluggish and difficult at the application level, spacy is smoother, quicker, and delivers a more efficient user experience. Additionally, it is ideal for *syntactic analysis* (which is a level of NLP discussed above in section 2.1.2). Unlike **Stanford CoreNLP** and other tools, Spacy integrates all functionalities, eliminating the need to manually choose modules. Additionally, the framework may be developed through the use of preset building components. Additionally, Spacy is great for performing *deep text analytics* and *sentiment analysis.* [13]

2.2 Textual Requirements Specifications and Their Problems

2.2.1 What is Requirements Specification?

Specification of Requirements is a subset of Requirements Engineering. [1] The major objective is to document the requirements for both the *user* and the *system.* Additionally, the requirements definition should be exact and easy to comprehend, which is challenging to achieve due to the diverse interpretations of stakeholders, resulting in *conflicts* and *inconsistencies* in the requirements. [14]

Specifications for requirements can be written as,

- Easily understandable text as a natural language specification

- Using a standard form or template in a structural manner.
- Create description languages by the use of programming languages.
- Using a Graphical notations and,
- The Mathematics specification requires the use of mathematical concepts such as finite-state machines or sets. On the other hand, a complete formal specification is difficult for customers or stakeholders to grasp.

Thus, among the aforementioned methods of constructing a requirements specification, a textual requirement written in natural language is the least complex. By default, it does not require any specified or standard format.

2.2.2 Problems in textual requirements

Both the client and the analyst can easily understand the requirements specification when they use a textual requirement. On the other hand, the shortcomings of textual requirements are [2]:

1. Ambiguity
2. Incompleteness and
3. Inconsistency

2.2.2.1 Ambiguity

Ambiguity refers to the ability of a word to be understood in two or more senses or ways. It is a frequent occurrence in natural language text. While establishing a *Requirements specification* in a textual format is more straightforward and intelligible, it is vital to discover ambiguity in the textual requirements. [15]:

Consider the following examples of ambiguity in sentences:

1. He runs the marathon.
2. She prepares dishes for dinner.

Sentence (1) creates ambiguity because it leaves open the possibility that *a person runs (organizes) a marathon competition* or *a person also runs (participates) in a marathon.* In the another statement, the same circumstance exists. It can be interpreted in two ways, *a man or woman preparing dishes (plates) for dinner.* Additionally, it suggests *she prepares dinner dishes (meal).*

Additionally, ambiguities are categorised according to their kind. They are listed and discussed below depending on the following:

- **Lexical Ambiguity**: It is defined as a term with several meanings. For instance, the term *black* refers to both *dark* and *corrupted (black economy)*. Lexical ambiguity also happens when two words have the same sound but have distinct spellings, such as *too* and *two*, *hole* and *whole*, and so on. [16]

- **Syntactic Ambiguity**: Additionally, it is referred to as *structural ambiguity*. This sort of ambiguity happens when a series of words contains words with varying grammatical structures, each of which conveys a distinct meaning. For instance, *Small toy store* may be interpreted in two ways: as (a small toy) shop or as a small (toy) shop. In one sense, the shop's size is smaller, while in the other, the toy's size is smaller. [16] Similarly, there are other statements that create syntactic uncertainty, for example, *He noticed the man with the field glass*, *The entrance near to stairs with the members-only sign* [3], and so on.

- **Semantic Ambiguity**: It occurs when a statement may be understood in more than one manner within its context, despite the absence of lexical or syntactic ambiguity. For instance, when several quantifiers appear in the same phrase, such as *all citizens have a personal identity number*, the statement can be understood in one of two ways: *each citizen has a personal identification number*, or *all citizens have a personal identification number.* [17]

- **Pragmatic Ambiguity**: This type of ambiguity happens when a statement has many interpretations within the circumstances of its production. For example, in a sentence *The trucks shall treat the roads before they freeze*, word *they* could mention both the roads as well as the trucks. [17]

- **Vagueness Ambiguity**: It occurs when a statement lacks the necessary meaning. For instance, *Software is a development platform for projects.* The term *platform* has many meanings in this statement. [16]

2.2.2.2 Incompleteness

An incompleteness problem occurs when certain associated information is missing or incomplete in a textual requirements specification. For instance, if a simplified model of a software system omits critical information essential for modeling a genuine system, Additionally, when stakeholders or domain experts collaborate with the analyst on textual requirement specifications, they will omit some details. [4] Taking the statement *Syntax error* as an example, it is not specified which program contains a syntax error. [16]

In a real-world scenario, if a functional requirement for an application that is capable of sorting the results is specified and is explained as follows: The

application should sort the results when the user views them, and it should sort the results according to *price, distance, restaurant name,* and so on.

According to the preceding explanation of textual requirements, a statement such as *sort the results, sort by price* omits critical information and generates an incompleteness issue, such as which results should be sorted. Moreover, at what cost?

2.2.2.3 Inconsistency

Textual requirements, in general, may not always convey consistent information about the system to be built. Because requirements may be produced by stakeholders with disparate interests, ambitions, and backgrounds, as well as limited subject expertise. [5]

There are two types of inconsistencies. One such instance is semantic overlaps between required clauses. For instance, if two conflicting assertions exist with the same subject and matching verb with object, this indicates that the statement and its negation are in the same condition concurrently. Additionally, inconsistency might occur as a result of requirement incompletion (omission of information).

Inconsistency will have a mixed effect. On the downside, it may delay and raise the cost of developing software systems, and there is no guarantee of safety or dependability. On the plus side, inconsistencies may aid in identifying elements of the system that require more examination. [18]

2.3 Linguistic Patterns

Linguistic patterns are grammatical principles that enable speakers and writers of a shared language to communicate effectively. Grammar, from the linguist's perspective, is not only a collection of rules; it also includes a set of blueprints that assist users toward producing explicit and predictable sentences. Languages and their divergence are based on grammatical structures. Additionally, all languages are composed of patterns that make sense of the language's properties, which include the arbitrary symbols, sounds, and words. [19]

2.3.1 TEMOS tool and Defined linguistic patterns

As previously stated, textual requirements include drawbacks such as *ambiguity, inconsistency*, and *incompleteness*. These mistakes must be corrected by a user, but they must first be brought to the user's notice. Thus, to address the issues of textual requirements, a tool called TEMOS (Textual Modelling System) was designed and implemented in the paper [2]. It assists in comparing portions of textual requirements specifications to corresponding fragments of a static UML model for possible inaccuracies. It is developed in the *Python*

programming language and makes use of the *Stanford Core NLP framework* for processing plain text in natural language using *annotators*. These are collections of processes for resolving various aspects of linguistics and generating notations describing the outcomes.

I shall describe in detail the TEMOS tool's primary components and the specified patterns, which are all relevant to this thesis work.

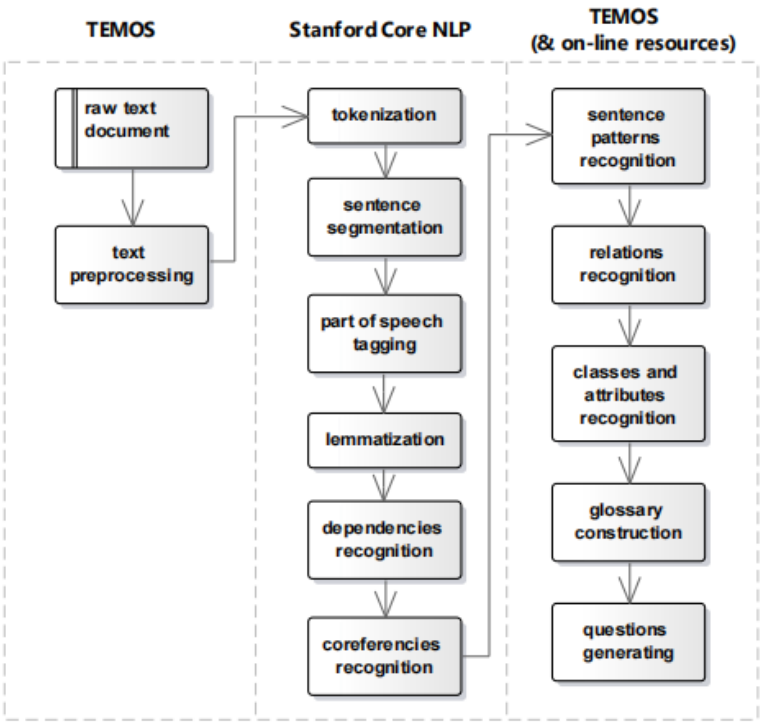

Figure 2.1: Workflow of TEMOS

[2]

Stanford Core NLP includes a number of annotation tools, however the TEMOS tool utilizes the following: [20]

- **Tokenize Annotator** - The input text was parsed and a lexical analysis performed using the tokenization annotator.
- **Sentence Annotation** (WordsToSentenceAnnotator) - Tokens are used to build sentences by *Words to Sentence Annotator* which splits a sequence of tokens into sentences.

- **Part Of speech Annotation** - It annotates tokens according to their POS tags. A Part Of Speech Tagger (POS Tagger) is a piece of software that scans text and assigns a part of speech to each word (token), such as a noun, verb, adjective, or other type of word. [21] Additionally, it annotates interpunction and other special characters in a text with the corresponding character. [2]

- **Lemma Annotation** (Morpha Annotator) - For each token, the *Morpha Annotator* creates a fundamental form (lemmas).

- **Dependency Parse Annotator** - This is the most remarkable annotation from the perspective of the TEMOS tool. It examines a sentence's grammatical structure and searches for links between words, such as the nominal subject of a verb and the dependent object of a verb. The direction of reliance is denoted by arrows, and each phrase has a root word that can be repeated one or more times and is not dependent on any input.

- **Coref Annotator**- This is the final annotator from *Stanford CoreNLP* that the TEMOS tool has utilized. This annotator's aim is to identify the words and pronouns it refers to.

TEMOS follows the *pattern-based approach.* The pattern-based recognition works by *grammatical inspection* of a word in a sentence, which has been done by the annotations. It largely depends on *dependency recognition* and *part of speech tagging* for this purpose, in order to determine the grammatical functions of words in textual requirements.

The pattern recognition algorithm then iterates over a sentence's *root words*, which correspond to a word's *part of speech tagging*, and, as indicated before, the most remarkable annotator, called the *dependency parse annotator*, establishes a dependence between words.

The result of the dependency parse annotator on a sentence is given in the below Fig 2.2.

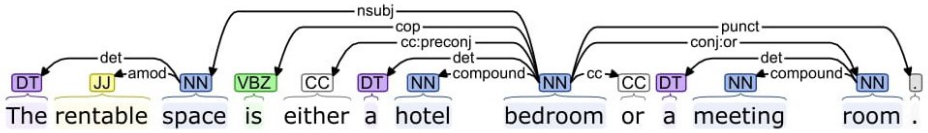

Figure 2.2: Result of Dependency parse annotator

[2]

Following the generation of a pattern based on the *dependency parse annotator*, the generated pattern will be compared to the *defined patterns* in order to identify a *class, attribute,* or *inside a sentence*. This is the fundamental concept implemented by a TEMOS tool, which converts the components of a textual document to a UML class diagram based on the annotations. Additionally, TEMOS developed several annotation types for the process of producing UML class diagrams. They are as follows:

- **Class Annotation** - similar to a *root*, this is a simple annotation that may exist on its own.
- **Attribute Annotation** - an annotation that is related with its owner or class annotation. For instance, if a *student* is a class, then its properties include its *name, age,* and so on.
- **Relation Annotation** - this annotation establishes a connection between the two classes. Thus, it would have both the *source* and *target* class annotations.

2.3.1.1 Defined Linguistic Patterns

The defined linguistic patterns are matched to the sentence, which is in a structured and lexical format immediately following an inspection of the grammatical roles, particularly those associated with the *part of speech tag* and *dependency parse annotation*. Thus, it is a pattern matching procedure that determines whether or not textual patterns exist in a given pattern. To convert a linguistically analyzed structure sentence to a UML diagram using the TEMOS tool, a sentence pattern must be matched to one of the provided patterns *Class/Attribute, Attribute pattern, General relation pattern,* or *Class Specialization pattern*. Thus, by comparing against these patterns, it is feasible to determine which word in a sentence is a class/attribute and how it is connected to other words (class). I've described each defined pattern and its graphic depiction in detail below based on [2]:

1. **Class/Attribute Sub-Pattern**: This sub pattern is utilized when a word matches the *class* or *attribute* annotations.

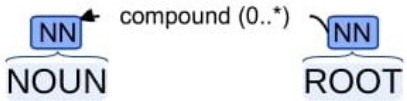

Figure 2.3: Class/Attribute Sub-Pattern

[2]

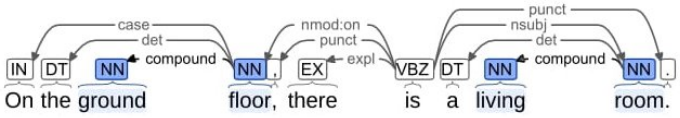

Figure 2.4: Class Attribute pattern matched sentence

[2]

Figure 2.4 illustrates the pattern matching of a sentence to a Class/Attribute pattern. The matching words are highlighted in the background.

2. **Class-Specialization Pattern**: This pattern is satisfying if some defined rules have been satisfied. They are:

   - In a sentence, the *root token* should be a *noun*. If this is the case, there will be a *class annotation*.
   - *Verb* (to be verb) should be a child with copula-type dependency (cop).
   - The second class annotation must exist as a *noun* and be a child of the root word with a *nominal subject*(nsubj) dependence.
   - If any *nouns* exist as children with *conjuct(conj)* dependency relations to the *root word*, they will be the other *class annotations*.
   - Finally, a relation annotation using the *source* and *target* classes would be produced. Additionally, *verb* denotes the class relationship.

   In Figure 2.5 and Figure 2.6, the Class specialization pattern and the sentence matched with it have been shown.

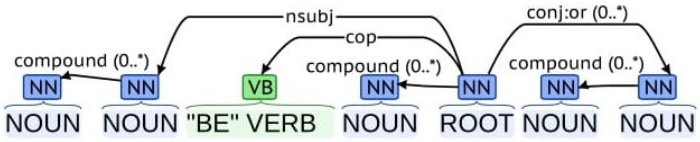

Figure 2.5: Class Specialization Pattern

[2]

3. **Attribute Pattern 1**: The purpose of this pattern is to extract *class* and *relationship annotation* attributes from textual requirements. When the following requirements are met, the sentence pattern will be matched.

   - A *root token* would be a *verb*, more specifically a *to have* or *to contain* verb.

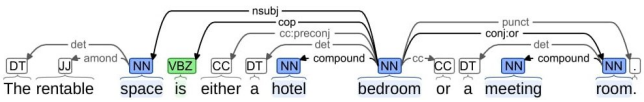

Figure 2.6: Class Specialization Pattern matching with sentence

[2]

- A *Class annotation* is a *noun* that must exist as a child of the type *nominal subject(nsubj)* dependence.
- Another *noun* must exist as a child of a *root word* through a *dependency object (dobj)*. Then this noun will be an annotation as an *attribute*. If any *noun* exists in the same way as this with a *dependence object* and as a child of the *root word*, it will likewise be annotated as *attributes*.

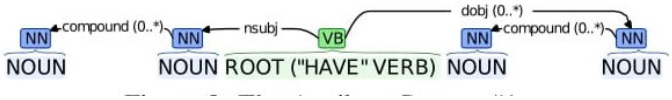

Figure 2.7: Attribute Pattern 1

[2]

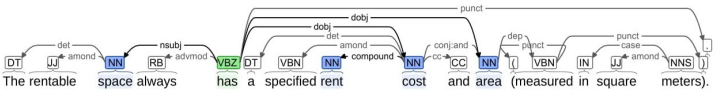

Figure 2.8: Attribute Pattern 1 matching with sentence

[2]

4. **Attribute Pattern 2**: This pattern indicates whether a root token is a *noun* or a *verb*, as indicated by the preceding attribute pattern. Additionally, it adheres to certain sets of regulations.

   - If the *Root token* is a *noun*, this will be a *attribute annotation (A1).*
   - Additionally, there must be a *noun* as a child of the *nominal modifier (nmod:of)* dependency type. If this is the case, a *class annotation(C1)* will be created.
   - If the *root token* has any other *noun* as a child with the dependency type *conjuction by and (conj:and)*, then the *attribute annotations (A2...An) will also be present.*

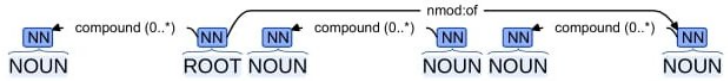

Figure 2.9: Attribute Pattern 2

[2]

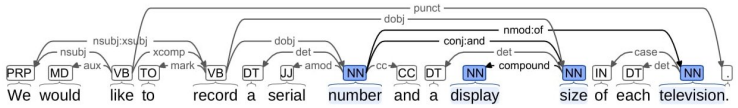

Figure 2.10: Attribute Pattern 2 matching with sentence

[2]

5. **General Relation Pattern**: This pattern can be used if none of the preceding ones match. Its construction is depicted in the diagram below. 2.11.

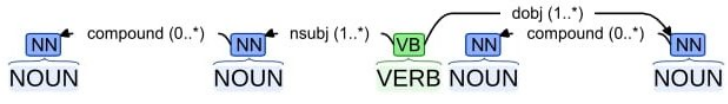

Figure 2.11: General relation pattern

[2]

Following that, sentences are matched against a predefined pattern. TEMOS generates UML models from classes and attributes. I will provide a brief overview of this procedure because it is beyond the scope of this thesis.

According to the paper [2], they created textual requirements for the hotel booking system and, using the *Class/Attribute, Attribute Patterns*, and other previously specified patterns, the TEMOS tool generated a UML diagram.

A few of the sentences that match the patterns are as follows:

- Our **business group owns** many **hotels.** The highlighted words are those that have the *Class/Attribute sub-pattern* and the *General relation pattern*, where **business group** and **hotels** are the two *class annotations* and **owns** is the *relation* between those classes.
- The **customer** is identifiable by the **name, surname, and address**; hence, this sentence matches the *Attribute pattern 1*, and the highlighted term **customer** is a *class*, and the **name, surname, and address** are *attributes* of the *class*.

Similarly, every other sentence will be matched to the defined patterns, and TEMOS will build a class diagram for each component, as seen in the following picture:

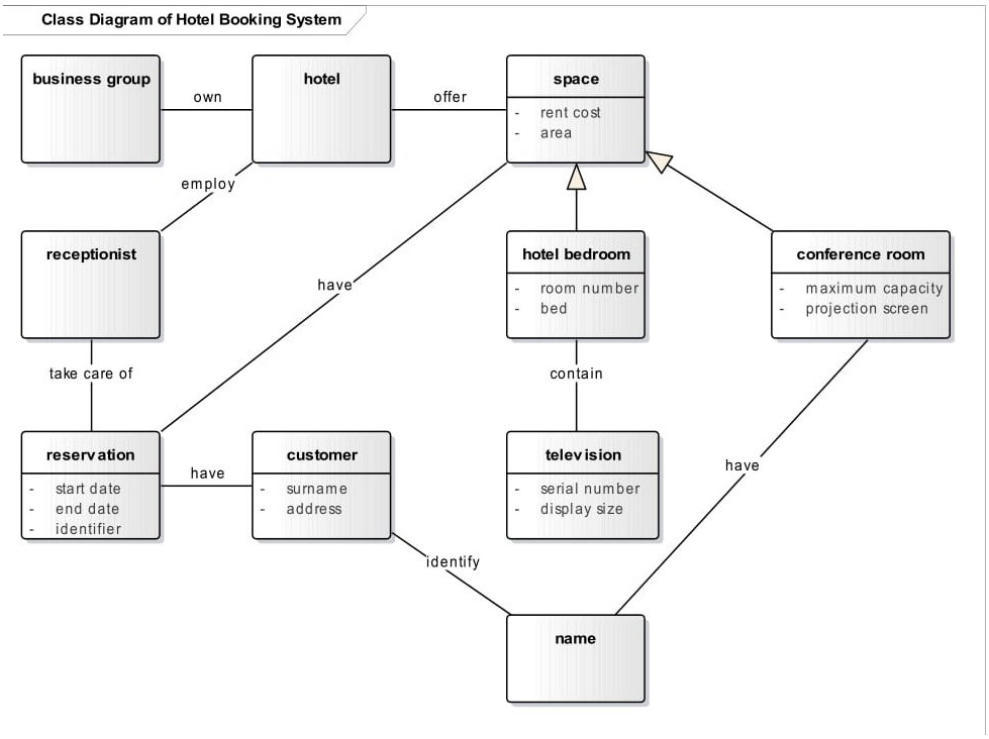

Figure 2.12: Generated Class Diagram

[2]

2.3.1.2 Linguistic patterns are used to identify the problems in a textual requirements:

The problems that occur in textual requirements, such as *ambiguity, incompleteness*, and *inconsistency*, as described in the section 2.2.2, are identified and resolved by matching against defined patterns introduced and designed in [3], [4], [5]. After implementing the pattern-based approach by matching a sentence against the patterns of *ambiguity, incompleteness*, and *inconsistency*, a TEMOS tool will generate warning messages to users writing textual requirements regarding the identification of inaccuracies and insistence on their resolution.

As a result, when employing a linguistic pattern matching technique to discover faults in a text, a *Neo4j* Graph Database should be used, as the patterns are represented as oriented graphs. Thus, the sentence to be analyzed

has been converted into a graph format based on its structural and lexical content by TEMOS using the framework *spacy-NLP*(section 2.1.3), and the graph has been saved and queried in a *Neo4j* database to get the matching patterns. [6]

In the next section, I'll discuss the Neo4j Graph Database and its features, benefits, and comparisons.

2.4 Graph Database and Neo4j

2.4.1 Graph Database

A graph database is a type of database that is meant to handle both the relationships between data and the significance assigned to the data itself. To save the data, it should first be diagrammed by illustrating how each unique entity is related to the others. [22]

2.4.1.1 Why Graph Database is efficient?

Everything is related in a real-world living environment. There are no separate pieces of information; everywhere around us, we perceive interrelated domains. Thus, only a database that is fundamentally relational is capable of effectively storing, processing, and querying connections. Additionally, while traditional databases calculate associations during the query process via complex JOIN operations, a graph database saves connections with the data in the model. Additionally, accessing nodes and relationships in a native graph database is a fast constant-time operation that enables rapid traversal of millions of interconnections per second per core.

A graph database is the most efficient way to manage densely linked data and sophisticated queries, regardless of the data's overall size. It utilizes simply a *pattern* and a set of starting points to explore the data around those initial starting points, collecting and aggregating information from millions of nodes and associations while leaving any material that is not relevant to the search undisturbed.

In comparison to other technologies, graph databases employ a variety of distinct methodologies, all of which are critical components of a graph database. One such technique is the *property graph model*, in which data is arranged as *nodes, relationships*, and *properties*. Data will be stored as *nodes* and *relationships*.

Nodes are the basic units of a graph, and they can have any number of attributes, which are expressed as key-value pairs called *properties*. *Labels* can be applied to nodes to indicate their various responsibilities within the domain. Additionally, node labels would deal with the attachment of meta-data (such as index or constraint information) to specific nodes. Meanwhile, *relationships* connect two node items in a directed, named, and semantically

meaningful way (for example, an employee WORKS_FOR a company, where WORKS_FOR is a relationship between nodes Employee and Company). A connection must always have an incoming or outgoing direction, a *start node* (the point at which the relationship begins), and an *end node*. Similarly, *nodes* and *relationships* can have *properties*, and in the majority of situations, these *properties* are quantitative, such as weights, prices, distances, and time intervals. Additionally, because to the efficient storing of connections, two nodes can share any number or kind of relationships without affecting performance. [22]

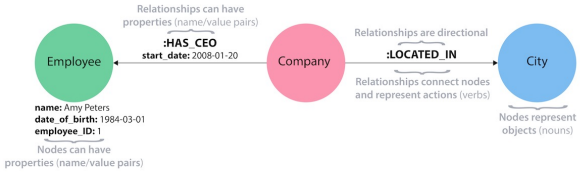

Figure 2.13: Property graph model

[22]

2.4.2 Neo4j

Neo4j is an open-source, NoSQL, native graph database that adheres to the ACID (Atomicity, Consistency, Isolation, and Durability) principles for application backend transactions. It was first created in 2003, including Java and Scala source code. Neo4j is available in two editions: Community Edition and Enterprise Edition. Additionally, it is referred to as a native graph database due to the efficiency with which the property graph model is implemented down to the storage level. This implies that data is stored precisely how we planned it, and the database navigates and traverses the graph using pointers.

Neo4j has complete database features, including ACID transaction support, cluster support, and runtime failover, which makes graphs appropriate for usage in production environments. [22]

Neo4j's popularity among developers, architects, and database administrators may be attributed to some of the following features:

- **Cypher**, a declarative query language comparable to SQL, but optimized for graphs.
- **Constant time intervals** for both depth and breadth in large graphs as a result of efficient node and relationship modeling. Additionally, it allows for a manageable expansion in the number of nodes to billions.

- **Flexible** property graphs that are changeable over time enable the development and addition of new relationships to progress the domain data as business requirements change.

- Neo4j offers **Drivers** to communicate with it in a variety of popular programming languages, including Java, Javascript,.NET, and Python.

2.4.2.1 Use Cases

Neo4j is extensively utilized by hundreds of businesses and organizations in almost every industry, including the following and several more. [23]:

- Detection and Analytics of Fraud

- Monitoring of network and database infrastructure for IT operations

- Social media platforms and graphs of social networks

- Management of identities and access

- Retail supply chain management

- Telecommunication

2.4.2.2 Neo4j vs RDF

Neo4j is related to the RDF (Resource Description Framework) technique that is frequently used for Linked Open Data. While both Neo4j and RDF representations may be visually expressed using *nodes* and *arcs*, the graph components are significantly different. In general, RDF is focused on the *edge-centered*, whereas graph databases are centered on the *node*. Additionally, graph databases would clearly distinguish between *properties* and *relationships*, and *labels* would be used to identify *nodes* and *relationships* depending on their entity (person, student, etc.), which is a notion that is similar to ontologies. [24]

While RDF may be queried using inference rules to extract implicit information from explicit relationships, Neo4j can be queried using the Cypher Query Language(CQL) to access nodes and their relationships to other nodes. Neo4j graph databases are better suited for traversing graphs and determining paths. In general, Neo4j has established a sophisticated and simple querying language called Cypher, which is built using ASCII graphical characters. I'll discuss the cypher query language in the next section.

2.4.3 Cypher Query

Cypher is a graph query language developed by Neo4j for the purpose of storing and retrieving data from a graph database. Neo4j's goal is to make graph data querying simple for anybody to learn, understand, and use, while simultaneously integrating the power and capability of other mainstream data access languages such as SQL. [25] Cypher's syntax provides a visual and logical means of matching node and relationship patterns in a graph. It is an ASCII-Art syntax-based SQL-inspired language for defining visual patterns in graphs. It enables users to write expressive and fast queries for graph data CRUD (Create, Read, Update, and Delete) actions. Additionally, it is not only the optimal method for interacting with data and neo4j, but it is also an open source initiative known as the *openCypher project*. [26] provides an open query language for property graph databases, a technical compatibility kit, and a *Cypher runtime*.

2.4.3.1 Why Cypher?

I previously discussed Neo4j's property graph model and its design of both *nodes* and *relationships* with associated properties in section 2.4.1.1. However, both nodes and relationships are basic components that contribute to the efficient and strong *pattern* property graph model. *Patterns* are a collection of *nodes* and *relationships* that may be used to visualize basic or complicated traversals and pathways across a graph. Additionally, the cypher is highly pattern-based and offers data in a graphical way. It is, in general, a pattern-based recognition system that communicates with users via visual diagrams. [27] As a result of the cypher's *pattern-based recognition*, it enables the storage and query of *linguistic patterns* in a *neo4j database*.

2.4.3.2 Representation of Nodes and Relationships in Cypher

The Cypher utilizes ASCII-Art patterns to graphically represent each component. [27] Thus, visual representation refers to *graphs*, and the primary components of a graph are typically *vertices* and *edges*. In neo4j, which makes use of the cypher query language, *vertices* are represented as *nodes*, and *edges* are used to describe *relationships* between the nodes.

For instance, I have created a sample *hospital* data set to give a brief explanation of how the cypher works. That data contains information about *patients, doctors*, and *relationship between patients and doctors.* The graphical representation of the neo4j has been given below in Figure 2.14:

What the represented graph says in a normal language is *Peter James(Patient) consults Richie Timoth(Doctor). JohnLeo(Patient) consults SindhuB(Doctor)* . And both patients are acquainted with one another.

The nodes in the data model can be identified by their nouns or objects. For example, in the represented graph, *Richie Timoth, Peter James, John*

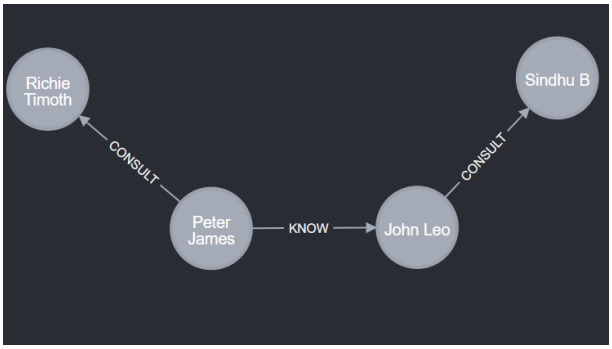

Figure 2.14: Graph with Nodes and Relationships

Leo and *SindhuB* are represented as nodes. The nodes in a Cypher query are written with brackets, e.g. (patient). Those parentheses look to show that the nodes are represented as circles in a visual data model.

- **Node Variables** - The variables relate to the nodes, and they can be accessed later through them. For example, variables (p) for patient and (d) for doctor node. It can be any variable name, exactly as variables in programming languages, and subsequent references can use the same name.
- **Node Labels** - Labels are node-specific tags that allow us to specify the sorts of things we can search for or create. For instance, the node labels will be *Patient, Doctor*, and so forth. This is comparable to SQL, which we may query to locate and obtain data from a certain table. Similarly, labels are used to do this in Cypher. As a result, if the node labels are not specified, Cypher will have difficulty checking all the nodes in the database. For instance, the following are some alternative methods for labeling nodes:

```
1 ()   //anonymous node (no label or variable)
2 (p1:PATIENT)      //using variable p1 and
3 label Patient for 1st patient
4 (p2: PATIENT)  // using variable p2 and
5  label Patient for 2nd patient
6 (:DOCTOR)   //no variable and labeled as Doctor
7 (d1:DOCTOR)  //using variable d1 and labeled as Doctor
```

- **Relationships between nodes** - By establishing relationships between nodes, the graph database becomes more efficient. Relationships are represented by an arrow, and depending on the incoming or outgoing relationship, we may use either right or left arrows between nodes. Although we establish relationships between nodes, without knowledge of what they contain, the relationship will be incomplete. Thus, the type

of relationship would be indicated as well, and it would be enclosed in square brackets. As an illustration, consider the following:

```
(p1)-[:KNOW]->(p2)
```

Thus, in this case, *p1* and *p2* are connected via the KNOW relationship type. Additionally, as seen in Figure 2.14, the relationship between nodes and the various sorts of relationships are described, including [:CONSULT] and [:KNOW].

Additionally, it is possible to describe a relationship just by dashes, without specifying either *incoming* or *outgoing* directions, implying that a relationship can be traversed in both ways. Therefore, if there is no directional relationship, the cypher will not perform in a certain direction and will instead obtain the relationship and all related nodes. This improvement to a cypher makes it more adaptable, rather than requiring users to be always aware of the direction of the connection.

- **Nodes and Relationship properties** - The property graph model's primary component is *properties*, which are name-value pairs that give more information about *nodes* and *relationships*. For example, the node PATIENT contains properties such as (name), (id), and so on, while the relationship type CONSULT includes a property called (problem). Additionally, the node's properties are queried.

```
(p1:PATIENT { id: "Pet", name: "Peter James",
address: "534 ErewhonPleasantville,Vic",
contact: "(03) 5555 6473", appointmenttime: "13:00"  } )
```

And *Relationship property* as,

```
-[c1:CONSULT { problem: "Cold" } ]->
```

2.4.3.3 CRUD(Create, Read, Update, and Delete) operations in Cypher query

- **Creation of Nodes and Relationships** - All *nodes* and *relationships*, as well as their associated properties, will be created through a clause called CREATE in the cypher. It can be constructed individually as nodes and then as relationships between them, or it can be constructed simultaneously as nodes and relationships. For instance, I generated the PATIENT and DOCTOR nodes in Cypher, along with their associated properties and relationships.

```
CREATE
  (p1:PATIENT { id: "Pet", name: "Peter James",
```

```
    address: "534 ErewhonPleasantville,Vic",
    contact: "(03) 5555 6473",
    appointmenttime: "13:00"  } ),
(p2:PATIENT { id: "John", name: "John Leo",
    address: "56Park st,LA",
    contact: "(03) 5555 6473",
    appointmenttime: "13:00" } ),
(d1:DOCTOR { id: "richie", name: "Richie Timoth",
             specialist:"General" } ),
(d2:DOCTOR { id: "sindhu", name: "Sindhu B",
             specialist:"Neuro" } ),
(p1)-[c1:CONSULT { problem: "Cold" } ]->(d1),
(p2)-[c2:CONSULT { problem: "Spine injury" } ]->(d2),
(p1)-[k1:KNOW]->(p2);
```

- **Reading the created or existing nodes and relationships** - The produced nodes and relationships can be retrieved using the MATCH and RETURN keywords. The MATCH keyword searches the neo4j database for an existing node, relationship, label, property, or pattern. This MATCH is analogous to the SELECT statement in SQL (Structured Query Language). Additionally, we can indicate which values or results we want to return from a Cypher query by using the RETURN keyword. While the return character is not necessary when constructing nodes or relationships, it must be present while reading. The node and relationship variables discussed previously are major elements when using the RETURN keyword to return values. [28]

  The following cypher query is to retrieve back *all the nodes and relationships* that have been created in the database as depicted in a Figure 2.14.

  ```
  MATCH (n) RETURN (n)
  ```

  To retrieve data with certain criteria, such as getting the id, name, and appointment time of a Patient who is consulting a specific Doctor:

  ```
  MATCH (p:PATIENT)-[:CONSULT]->(d:DOCTOR
  {name:"Sindhu B"})
  RETURN p.id,p.name,p.appointmenttime
  ```

  The above cypher query will give the results as shown below in Figure 2.15.

  In addition, it is also possible to *filter the query* results using WHERE condition [29] and also sort the values either by ascending or descending order using the keyword ORDER BY. [30]

Figure 2.15: Returning the specific values

Example:

```
//Return the name of a doctor who is a specialist in General
MATCH (d1:DOCTOR)
WHERE d1.specialist = "General"
RETURN d1.name;
```

- **Updation in Cypher** - Cypher executes update operations by using the SET keyword to modify the node's properties. To begin, we can use a MATCH to locate the node for which we need to make changes, and then, using the SET keyword (with the syntax *variable.property*), we can *add, delete, or update* a node's property, such as its *name, age,* and so on. [31]

```
//Adding a new property contact to a doctor.
MATCH (d:DOCTOR {name:"Richie Timoth"})
SET d.contact = "rt11@gmail.com"
RETURN d.contact
```

- **Deletion of nodes and relationships** - Cypher includes the DELETE keyword, which allows to delete *nodes* and *relationships* from a database. However, because Neo4j is an ACID-compliant database, deleting a node that is connected to other nodes is not possible. Because if it were conceivable, it may have resulted in a *relationship pointing to nothing*, resulting in a *incomplete graph.* Thus, the relationship should be destroyed first, followed by the nodes.

2.4.3.4 Cypher query vs SQL

Anyone familiar with SQL may build cypher statements for graph database management using cypher queries. Due to the fact that it is similar to SQL in

terms of declarative and textual query languages, but is designed for graphs. [32]

The Cypher is composed of several commonly used SQL clauses, keywords, and expressions (predicates and functions), including WHERE, ORDER BY, SKIP, LIMIT, and AND. Meanwhile, Cypher describes graph patterns by utilizing the MATCH keyword to find the exact pattern in a database and returning it using the RETURN keyword. This process is comparable to SQL's SELECT and FROM keywords, which are used to get records from a *table*.

For example, to display the entire contents of the *students* table in SQL. It can be queried as,

```
SELECT s.*
FROM student as s;
```

Similarly, in cypher, it can be done by matching a pattern that returns all the nodes labeled as :STUDENT.

```
MATCH (s:STUDENT)
RETURN s;
```

Furthermore, *sorting* results using the ORDER BY clause and filtering results based on equality using the WHERE condition are both supported in cypher, just as they are in SQL.

Additionally, Cypher provides a relationship between nodes, analogous to SQL's JOIN operations. For instance, the following SQL and Cypher code can be used to do JOIN operations between a teacher and a student, as well as to extract the teacher's name:

JOIN operation in SQL:

```
SELECT t.Name
FROM Teacher AS t
JOIN Student AS s ON (s.StudentID = t.TeacherID)
```

Relationship in Cypher

```
MATCH (s:Student)<-[:Teaches]-(t:Teacher)
RETURN t.name
```

Similarly, Cypher supports a variety of JOIN operations through RELATIONSHIP. Additionally, other SQL keywords, such as

- DISTINCT(to return distinct results)
- GROUP BY (Grouping)
- COUNT, SUM, AVG, MAX (Aggregations)

2.4.3.5 Procedures and Functions

Apart from providing an efficient graph database experience, Neo4j and Cypher are occasionally required to execute more advanced features like as parallelization, additional graphical algorithms, and so on. As a result, an expansion of Neo4j and Cypher with *User-Defined Procedures* and *Functions* was required for that purpose. [33]

What procedures and functions does this section refer to?

Indeed, because the *functions* perform fundamental computations and return a single value, they are applicable to every expression or predicate. On the other hand, *procedures* carry out intricate operations and generate streams of output. Additionally, it must be used within the CALL statement of Cypher, with the results accessible via the YIELD keyword. Moreover, they can generate, obtain, or calculate data for later Cypher query processing stages.

These functions and procedures are included in Neo4j and can be accessed through the following statements in a Neo4j browser.

```
This is to display all the available procedures.
CALL dbms. procedures()

This statement is to display all available functions.
CALL dbms. functions()
```

APOC Library:

Neo4j laboratories is a collection of the most cutting-edge developments in graph technology, including an incredible collection of libraries developed by the Neo4j community and team. The most extensive toolkit among those projects is the *APOC* (Awesome Procedures on Cypher) library, which includes a variety of procedures and functions (around 450) for data integration, graph remodelling, data transformation, operational functionality and so on. [34]

Some of the procedures and functions included in that huge variety are as follows: [35]

- **apoc.create** This procedure assists in the dynamic *creation* of nodes, relationships, properties, and labels, as well as many other actions.
- **apoc.merge** This procedure is used to dynamically conduct a *merging* operation. Actually, the MERGE operation is almost identical to the Cypher keyword CREATE, except that it adds nodes and relationships only if they do not already exist in the database, hence avoiding duplication.

Similarly, the *APOC* library has additional procedures and functions for dealing with natural language text, doing mathematical calculations, and executing other cypher queries. Additionally, I will discuss the precise *APOC*

procedures and functions that will be related to and used to accomplish my dissertation in my subsequent analysis.

2.5 Summary

In summary, this chapter introduced Natural Language Processing (NLP) by describing its various levels and methods for processing text into a structural and lexical format, as described in section 2.1.

Following that, the explanation of textual requirements specifications was discussed, as well as the inaccuracies of that in section 2.2.

Following that, in section 2.3, a thorough description of linguistic patterns and the background for this thesis are offered, as well as a brief introduction to the TEMOS tool and its capabilities.

Finally, I introduced the Neo4j graph database and its operations using the Cypher Query Language (CQL) in section 2.4. These were demonstrated through the use of CRUD (Create, Read, Update, and Delete) activities. Additionally, I compared the Cypher query language to existing database query languages. Additionally, I discussed the sophisticated functions and procedures available in Neo4j.

Thus, the fundamental knowledge necessary for this thesis is presented. Now is the time to do an analysis of the resources required to commence implementation.

Chapter 3

ANALYSIS AND DESIGN

This chapter will detail the analysis I undertook to determine the best method for achieving my thesis's objective. It will explore the tools required to construct an interface between *Python* and *Neo4j* that will operate as a common interface with TEMOS, allowing for the extraction of Query(sentence patterns) and pattern matching in *Neo4j*. Certainly, this chapter will cover the context in which I received the concepts for performing a match between the pattern of a Query (sentence) to be analyzed and the defined linguistic patterns.

Following the study, a design will be created for a *Neo4j* graph database to store and conduct pattern matching, as well as an API interface between TEMOS and Neo4j.

3.1 How to create an interface with Neo4j?

I began my investigation by attempting to determine how to construct an interface with Neo4j. Additionally, I discovered that Neo4j supports the *.Net, Java, JavaScript, Go,* and *Python programming languages* via the binary *Bolt protocol*. However, Neo4j's community drivers support all programming languages, protocols and APIs. [36] Thus, it is important to connect to Neo4j using the desired programming language for developing an application, and only then can the database be accessed. Without a doubt, I prefer to utilize Neo4j in conjunction with Python. Additionally, the Python interface was chosen since the TEMOS tool (section 2.3.1) is built in the Python programming language. Thus, if I query Neo4j from Python through an interface, I will have access to a TEMOS tool and will be able to receive a Query (sentence pattern) for analysis and then compare it to a Neo4j database of saved linguistic patterns to obtain results.

Additionally, it is hassle-free to utilize a driver that connects to Neo4j through either the Binary *Bolt protocol* or *HTTP*.

3.1.1 How the binary Bolt Protocol works in Neo4j Python driver?

Neo4j by default employs the *Bolt binary protocol*, which is a *client-server* protocol optimized for database applications. It is more efficient and lighter than the previous method. *Bolt* protocol versions run via TCP or Websocket connections with optional TLS encapsulation. This protocol is statement-oriented and enables clients to transmit statements composed of a single string and a set of typed parameters. The server will respond to all requests with a result message and an optional stream of result records. [37]

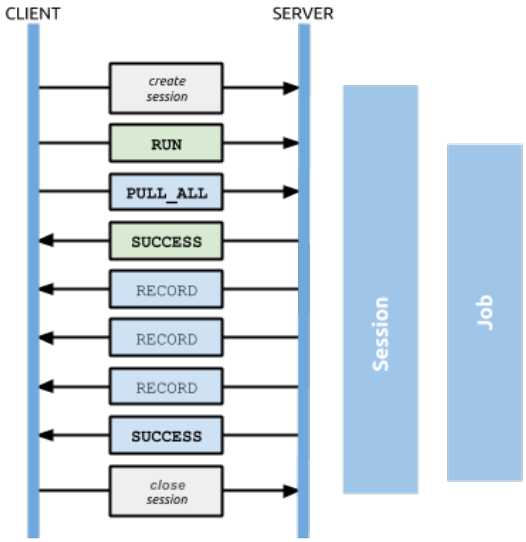

Figure 3.1: Workflow of Bolt protocol

[37]

I analyzed the bolt protocol in Python by installing a *Neo4j Python driver* in order to understand how the database is accessed, and I've summarized the method below along with code snippets:

- **Connection URI and Authentication**: I have started a connection using a bolt protocol URI and user credentials to access the database. [38]

```
uri = "bolt://localhost:7687"
userName = "neo4j"
password = "123"
driver = GraphDatabase.driver(uri, auth=(userName, password))
```

- **Session and Transactions**: After authentication, a new *session* is generated between the *client* and a *Neo4j server*, and this session enables the user to begin a transaction by running a cypher query statement that should be executed. Additionally, can commit those transactions to a database. Actually, transactions are the units of work (jobs) executed throughout a session. [39]

  For instance, the following code sends the Cypher query as an auto-commit transaction. In this manner, a single query statement can be sent per transaction. Additionally, it will not retry in the event of a failure. [40] For instance, the following code simply establishes a session with a database and executes a query to retrieve the doctor's name before storing the result as a record (list) and printing it. It is only capable of completing a single instance in this circumstance.

```
with driver.session() as session:
    result = []
    matched = session.run("MATCH (d:DOCTOR) "
                          "RETURN d.name AS name")

    for record in matched:
        result.append(record["name"])
    print(result)
```

  On the other hand, I am able to perform a query for two distinct instances by utilizing the transaction procedures. For instance, the following code executes (sends) a query to retrieve the name of a patient who has a relationship [: CONSULT] with a certain doctor. Additionally, a WHERE condition is used to filter the doctor's name. Additionally, the session has two read transactions, each of which contains a single instance, i.e., the doctor's name, which serves as the input for the WHERE condition's value. The transaction was then repeated several times to satisfy both instances and put the results into a record, and then I obtained all the remaining result streams using a PULL ALL request. Indeed, PULL ALL is the action of obtaining from a server all remaining results as *records*. If the request is successful, the server will provide the results.

```
def get_patient_of(tx, name):
    patient = []
    result = tx.run("MATCH (p:PATIENT)-[:CONSULT]->(d:DOCTOR) "
                    "WHERE d.name = $name "
                    "RETURN p.name AS patient", name=name)
    for record in result:
        patient.append(record["patient"])
```

```
        return patient


    with driver.session() as session:
        patient = session.read_transaction\
            (get_patient_of, "Richie Timoth")
        for patient in patient:
            print(patient)

    with driver.session() as session:
        patient = session.read_transaction\
            (get_patient_of, "Sindhu B")
        for patient in patient:
            print(patient)
```

As a result of this brief examination, I learned about the creation and management of interfaces and transactions with a Neo4j database using a *binary bolt protocol* and a *Neo4j python driver*.

3.1.2 HTTP API

Neo4j is also available through an HTTP API. We may submit a POST request to Neo4j that contains a JSON representation of the cypher query. It leaves the transactions open for many requests until we *commit* or *rollback* the modifications. After the request is approved, the response will be given in the form of JSON streams including result items. [41]

To begin, the transaction must be initiated by submitting a POST request to the transaction endpoint containing zero or more cypher queries. The server then answers to this request with the result elements for the cypher query as well as the transaction id associated with it. Following that, we can include further cypher statements and *commit* the transactions to the database by referencing the transaction's location. Additionally, the DELETE method enables the *rollback* operation to delete the requests, and any subsequent statements will fail instantly.

Additionally, it is feasible to *begin* and *commit* a transaction within a single HTTP request.

I used the *Postman API* tool to test the HTTP API transaction requests described in Figure 3.2 with my Neo4j database.

What is Postman API ?

Postman is an API development and consumption platform. The postman tool assists in the design, testing, and discovery of APIs. [43]. I used this tool to determine how Neo4j is available via the HTTP API, in the hope that

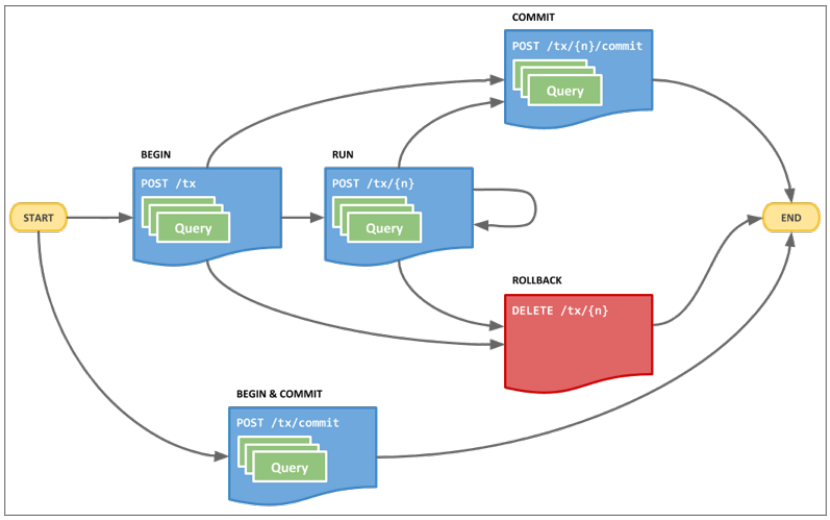

Figure 3.2: HTTP API Transaction flow

[42]

it would provide me with some insight into how the interface between Neo4j and HTTP may be implemented.

It is important to authenticate and approve the HTTP requests that will be sent before connecting to Neo4j. Thus, it is possible by supplying the right login and password for a Neo4j database that has been built.

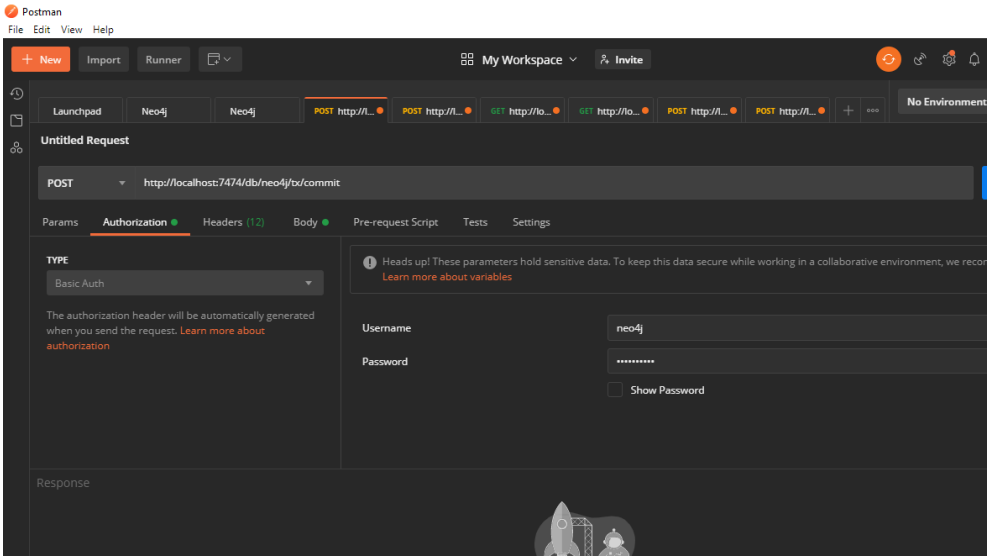

Figure 3.3: Screenshot for Neo4j Authorization

On a single HTTP request, I *initiated* the transaction and also conducted a *commit* operation.

And I submitted the POST request as specified below, along with JSON statements containing a Cypher query for node creation.

- POST http://localhost:7474/db/neo4j/tx/commit
- Accept: application/json;charset=UTF-8
- Content-Type: application/json

JSON statement with query:

```
{
 "statements": [
  {
  "statement": "CREATE (n:Person {firstName: $name}) RETURN n",
        "parameters": {
                "name": "KK"
            }
  }

    ]
}
```

As a response, I received a status confirmation and a JSON result element.

- 200: OK
- Content-Type: application/json;charset=utf-8

```
{
    "results": [
        {
            "columns": [
                "n"
            ],
            "data": [
                {
                    "row": [
                        {
                            "firstName": "KK"
                        }
                    ],
                    "meta": [
                        {
```

```
                        "id": 0,
                        "type": "node",
                        "deleted": false
                    }
                ]
            }
        ]
    }
],
"errors": []
}
```


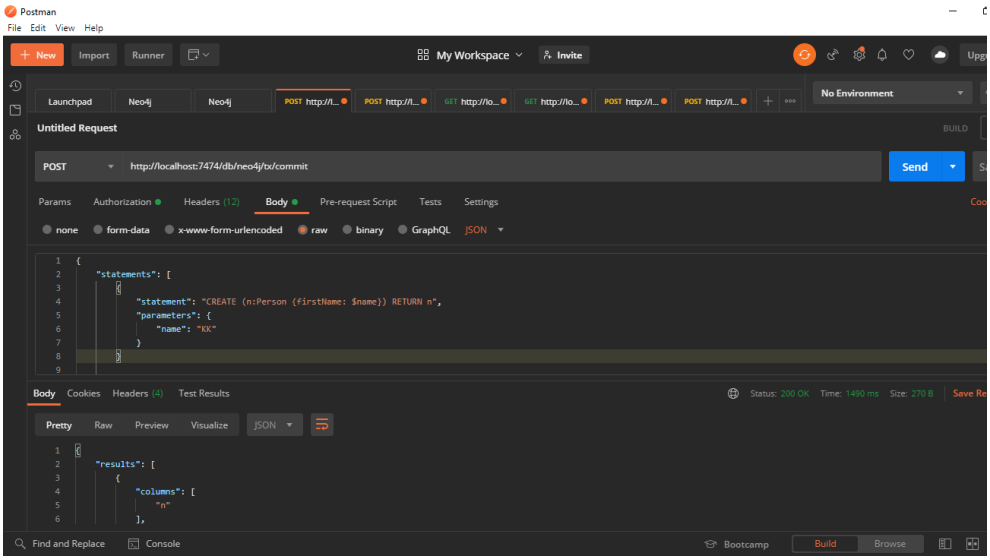

Figure 3.4: Sending HTTP request from Postman API tool

As a consequence, the node was built based on the sent cypher query. I have added a screenshot of it in Figure 3.5.

After examining the Bolt protocol and HTTP API, I acquired a better understanding of how the Neo4j interface works, which led me to pick one of Neo4j's community drivers, called **Py2neo**. I'll discuss the tool's features and why I chose it in the next section.

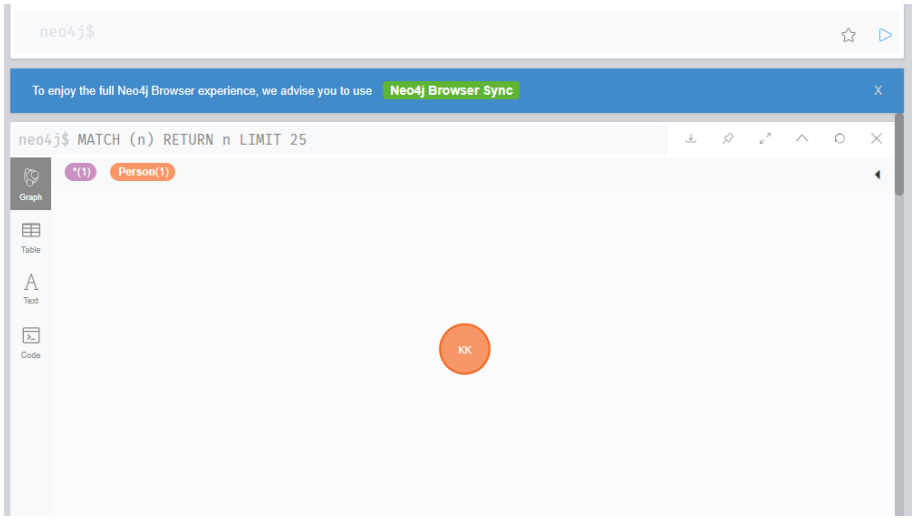

Figure 3.5: Node creation in Neo4j Database

3.2 Decision to use Py2neo community driver for Neo4j interface

After analyzing the Binary Bolt protocol's behavior in the native Neo4j driver and the HTTP API transactions, I decided to use the Neo4j community driver to create an interface between *Python* and *Neo4j*

What is Neo4j community driver? Community drivers are created by members of each programming language community to facilitate communication between the chosen programming language and the Neo4j database. It is available for the majority of programming languages, including C, Python, Perl, and PHP. [36] Considering the requirement for a Python interface, I focused on the Neo4j community driver for Python.

Python community driver - The available community drivers for python are:

- **Py2Neo** - It is a client library and a more extensive toolkit that enables users to interact with Neo4j from Python programs or the command line. [36]
- **Neomodel** - The Neomodel driver, on the other hand, is an Object Graph Mapper that supports Django models. Django is a Python web framework for developing web applications that use the Neo4j database as a backend. [44] So, picking this driver will be irrelevant, as I will not be working with any web frameworks.

Hence, I chose Py2Neo as my preferred driver for creating an interface with the Neo4j database.

3.2.0.1 Py2Neo

As previously stated, Py2Neo is a library that enables client-side Python programs to communicate with Neo4j. *Nigel Small*, a Neo4j API specialist, created it. Additionally, this library supports both the *Bolt protocol* (section 3.1.1) and the *HTTP API* (section 3.1.2) and provides a high-level API experience for Neo4j. This API's core section is a *Graph API*. It has a Graph class that represents a graph database exposed by a Neo4j server running on a single instance or in a cluster, and it offers access to a substantial amount of the py2neo capabilities. Additionally, the *GraphService* object represents the entire graph database system. [45]

Additionally, *py2neo* has built-in methods and classes for implementing certain functionality. All of them are kept in the *py2neo root namespace*. It is like a huge module, and it contains the codes that define each function and class.

The features of these methods and classes, which include database connection, database administration (storing and retrieving nodes and relationships from the database), and error handling. I'll describe Py2neo's functionality and process in the following parts.

3.2.0.2 Connection

To begin with, it establishes a connection to the database. Generally, py2neo makes extensive use of the predefined classes *Graph* and *GraphService*, which both aid in creating the groundwork for accessing the Neo4j database. Additionally, these classes receive the profile, which is actually a URI (Uniform Resource Identifier), and any other unique settings during the API's development. Both will define how and where a connection is possible. [46]

The URI profile conforms to the standard format, which is as follows:

```
<scheme>://[<user>[:<password>]@]<host>[:<port>]
```

Some of the URI schemes (protocols) that are supported are as follows: :

- **neo4j** - Bolt with routing(unsecured)
- **neo4j+s** - Bolt with routing(secured with certificate checks)
- **neo4j+ssc** - Bolt with routing(secured but no certificate checks)
- **bolt** - Bolt direct connection (unsecured connection)

- **bolt+s** - Bolt direct(secured with full certificate checks)
- **bolt+ssc** - Bolt direct(secured with no certificate checks)
- **http** - HTTP direct (unsecured)
- **https** - HTTP direct (secured with full certificate checks)
- **http+s** - HTTP direct(secured with full certificate checks)
- **http+ssc** - HTTP direct(secured with no certificate checks)

Following are the individual settings needed for the connection:

- **uri** - A complete *profile URI* that is supplied as an argument to the *Graph()* constructor.
- **scheme** - Any of the above-mentioned supported schemes may be utilized.
- **protocol** - It specifies the protocol name that will be used for communication, for example, *bolt* or *http*.
- **address** - Host name and the port number of the Neo4j server to make a connection. For example, the address can be represented as *localhost:7687*.
- **authentication** - The database's authentication details include *username* and *password* to establish a connection.
- **secure** - If a *flag* is used to indicate that a connection should be secured, the connection will be secured using a TLS (Transport Layer Security) certificate, commonly known as a Socket Secure Layer (SSL), which may be accomplished with Python's built-in *ssl module*.
- **verify** - This is a *flag* indicating that the server certificate should be checked thoroughly. However, this requirement applies when the preceding *secure setting* is met. That is, this only applies if the connection is secure.
- **routing**- A *routing flag* has been utilized if the connections should be routed through various servers. This feature, however, is available only in *service profiles*, not in ordinary *connection profiles*.

This is how the *Graph()* class from the *py2neo* module was exported. Additionally, the Graph constructor receives an argument specifying the URI, protocol, address, and authentication credentials. And finally, if the details are valid, the connection to the Neo4j database is feasible.

```
from py2neo import Graph
g = Graph("bolt://localhost:7687", auth=("neo4j", "123"))
```

Profile objects for Connection

The *py2neo* module includes several pre-defined profile objects, including the *ConnectionProfile* and ServiceProfile objects for connection. These objects are used when the URI, authentication information, and other configuration parameters are left undefined. [46]

For instance, if the *Graph()* constructor is empty and has no parameters. Then, using default settings, the bolt protocol was used to connect to a *localhost* on port 7687, with the default username and password being *neo4j* and *password*.

```
from py2neo import Graph
g = Graph()
```

The *ConnectionProfile* contains default values for establishing a connection to the Neo4j database, as well as the necessary individual parameters that are set as default and can be modified via environment variables.

Assigned default values are:

```
DEFAULT_PROTOCOL = "bolt"
DEFAULT_SECURE = False
DEFAULT_VERIFY = True
DEFAULT_USER = "neo4j"
DEFAULT_PASSWORD = "password"
DEFAULT_HOST = "localhost"
DEFAULT_BOLT_PORT = 7687
DEFAULT_HTTP_PORT = 7474
DEFAULT_HTTPS_PORT = 7473
```

Available Environment variables:

```
Neo4J_URI
NEO4J_AUTH
NEO4J_SECURE
NEO4J_VERIFY
```

How it works?

Each setting is produced as a *instance attribute* with its associated default values. Following that, the property function manages those attributes, which is signified by the *@property decorator*. Additionally, distinct functions are constructed to specify how each action should behave. These routines verify

that the connection's needed settings, such as URI, protocol, and authentication information, are met; if they are not, an exception is thrown. Additionally, if no values are supplied and are unaffected by environment variables, the default values assigned will be utilized.

For example, if no *uri* is specified, the default bolt URI is used. Similarly, if none of the other individual options are supplied, they will be set to default. However, if the user has configured the graph database's password to a default password, the connection will succeed. Otherwise, authentication will fail and no connection to the database would be permitted. I examined this by just running a connection test from the command line in Python. The py2neo module code for ConnectionProfile is presented in [47].

The *ServiceProfile* provides the same connection details as the *ConnectionProfile*, but it is for a full Neo4j service, such as cluster or single instance management. Additionally, all of *ConnectionProfile's* characteristics have been inherited. Additionally, whenever the profile defines connectivity to a full Neo4j service, the *routing option* is provided. However, unlike other properties, this routing option is not activated by default.

3.2.0.3 Database Management

Graph Service objects- Py2neo manages databases using predefined *classes* and *objects*. The most fundamental class, called *Graph Service*, comprises graph objects used for storing and retrieving activity. [48]

The URI can be passed to the GraphService() constructor as well:

```
from py2neo import GraphService
gs = GraphService("bolt://localhost:7687")
```

Alternatively, no parameters can be supplied to the constructor. The default URI *bolt:/localhost:7687* will be considered in that case.

```
gs = GraphService()
```

Additionally, the GraphService class has some functions (which include property functions) and attributes that enable certain operations to be performed. They are as follows:

- **iter(graph_service)**

  This method returns a list of all named graphs in the database.

- **graph_service[name]**

  This method provides access to a Graph class.

- *property* **config**

  A dictionary has been created to store the configuration parameters necessary to configure Neo4j.

Likewise, additional methods are offered for creating a *connection*, obtaining a *kernel version* of the running Neo4j program, and creating a shortcut for *ConnectionProfile*.

Graph objects

This *Graph()* class is essential in Py2Neo. It accomplishes it by administering a named graph database that is available via the Neo4j graph database service. Naturally, the connection information, as well as any relevant specific settings, can be given by URI or Connection Profile. Indeed, the *name* parameter in this Graph class enables the graph database to be identified by its *name*. Additionally, passing a *None* for the name parameter selects the default database that the user starts and runs on the Neo4j server. [49]

Additionally, the *SystemGraph* class can manage the graph database. It is a subclass of the *Graph* class and enables remote DBMS access to the system database (Database Management System).

Once the connection is established, the *Graph* class offers a means of accessing the majority of the functionality offered in *py2neo*. This function enables you to handle both the graph database and the Cypher query. Indeed, these functions implement the CRUD operations of the cypher in a *pythonic* fashion (as explained in section 2.4.3.3). Additionally, transactions (as indicated in section 3.1.1) will be formed in the same manner as the native Neo4j driver, allowing for the simultaneous execution of one or more graph operations. *auto* and *begin* methods generate and terminate these transaction objects, respectively. *commit* and *rollback* methods commit and rollback these transaction objects. [50]

Execution of Cypher Queries - Direct Cypher queries can be conducted and transmitted to the database within the transaction. To do this, *py2neo* provides techniques such as,

- **evaluate()** - This function conducts a single cypher query and returns the value of the first record's first column. [51]

- **run()** - The cypher query can be sent to the Neo4j server for execution, and the results are returned as a stream of *records*. [52] Indeed, *record* is more akin to a list that has an ordered, keyed collection of data. By giving their *key* or *index*, the values contained in that can be retrieved. Additionally, a *Cursor* object is utilized to browse the record stream.

- **update()** - A function that does a single cypher query but does not deliver any results. For example, it modifies the database by adding nodes or relationships but does not return any results.

Apart from directly executing cypher statements using the aforementioned methods, Py2neo supports CRUD actions via *Subgraph* objects. In general, a *Subgraph* is a collection of random nodes and connections; it also serves as the basis class for the *Node, Relationship*, and *Path* classes. By integrating the nodes and relationships, a subgraph may be formed. Additionally, the rationale for calling it *Subgraph* is because it aggregates all of py2neo's data values and then uses them as arguments for a variety of graph database operations, such as Graph.create(), Graph.merge(), and so on. This technique enables the efficient transmission of several items to the database in a single round trip, while also lowering network overhead. [53]

The different *Subgraph* operations are:

- **create(subgraph)** *Node* and *Relationship* objects are created by assigning them to a local *Subgraph* and then sending them as inputs to the constructor to make modifications (create) in a database.

  For example, the following code creates two nodes using the *Node* and *Relationship* objects and assigns them to the variables *a* and *b*. Following that, it is supplied as a *Subgraph* parameter to the g.create() function during database construction.

```
g = Graph("bolt://localhost:7687",
          auth=("neo4j", "123"))
a = Node("Patient", name="John")
b = Node("Doctor", name="Richard")
ab = Relationship(a, "CONSULT", b)
g.create(ab)
```

  By experimenting with the above code, I am able to see nodes and relationships that have been created in my Neo4j database, as depicted in Figure 3.6.

- **merge(subgraph)** A subgraph *merging* operation creates or updates a subgraph's *nodes* and *relationships*. It is accomplished by comparing the node to the database's existing nodes. If it matches the node's label and property value, no new node is formed. Rather than that, if any further property values are mentioned, the node will be changed. In the absence of an exact match, the merge operation will produce a new node. The same follows for relationships in which both the start and end nodes, as well as the relationship type, have been compared. If no

Figure 3.6: Graph creation by create() method

match is discovered, a new relationship is made; if a match is found, the existing one is modified. Thus, the merging procedure is analogous to avoiding duplicate nodes and linkages.

merge(subgraph, primary_label=None, primary_key=None)

Parameters of merge function:

- **subgraph** - a Node, Relationship or other Subgraph object can be given.
- **primary_label** - a label which wants to be checked for it's existence in a database.
- **primary_key** - property key(s) of the nodes to be checked for existence.

```
from py2neo import Graph, Node, Relationship

g = Graph("bolt://localhost:7687",
          auth=("neo4j", "123"))
a = Node("Patient", name="John")
g.merge(a, "Patient", "name")
b = Node("Doctor", name="Richard")
g.merge(b, "Doctor", "name")
ab = Relationship(a, "CONSULT", b)
g.merge(ab, "Patient", "name")
```

In the above example, each node and relationship in the subgraph have been merged separately. Besides, the *label* (Patient and Doctor) and *property key* (name) are used for the comparison with the existing nodes and relationships in the database.

- **delete(subgraph)** This function may be used to remove distant nodes and relationships that correspond to the local subgraph. If, on the other hand, just the relationships are to be deleted, the *separate()* function can be utilized.

- **exists(subgraph)** It is used to determine whether or not a graph contains one or more entities.

- **pull(subgraph)** A *pull operation* is used to update the entities (nodes and relationships) in the local subgraph from their distant equivalents.

- **push(subgraph)** Push is the inverse of pull, which updates remote entities from their local counterparts.

Following the storage of entities in the remote Neo4j database, it is essential to read or return the execution results. In that regard, the MATCH keyword can be used in conjunction with either direct cypher execution or the py2neo-provided methods for matching nodes and relationships. Additionally, py2neo supports certain criteria for sorting and filtering results via ORDER BY, WHERE, LIMIT, SKIP, COUNT, and a variety of other functions analogous to Cypher clauses. [54] Additionally, several forms of problems occur in Neo4j and during its connection. This may be determined using py2neo's *Error messages*. They are accessible for *ClientError, DatabaseError*, and other *Connection-related errors*. [55]

Eventually, from the overall analysis of py2neo and its various operations, I came to know about how the Neo4j database can be managed by executing *cypher* queries from *Python* programming language using py2neo as an interface, and also by using a plethora of functions and objects available in *py2neo*. So, this analysis on *py2neo* leads to the efficient way of storing the defined linguistic patterns in a Neo4j database. This is one of the objectives of my dissertation. Moreover, this Python interface makes it possible to interact with the TEMOS tool, as it is written in the Python programming language. Therefore, I can obtain the *sentence* to be analysed as an entry in the Neo4j database. Finally, after successfully storing the input sentence patterns from TEMOS and predefined linguistic patterns, I am able to match those patterns to achieve my goal for this thesis.

3.3 Analysis of linguistic pattern matching in Neo4j

Followed by an analysis of the Python interface for the Neo4j database, it is necessary to find a way to get the Query (sentence) from TEMOS as an input to the Neo4j database and implement the pattern matching. So in this section

I'm going to do my analysis of that. First, it begins with finding a better way to receive the sentence patterns as an entry in the Neo4j database.

3.3.1 How can I obtain the sentence from TEMOS as an entry in the Neo4j database?

In accordance with the requirements, I must create a method as a shared interface. By calling that method, TEMOS should be able to send the sentence as a *list*, which holds grammatically inspected sentences based on annotations such as part of speech tagging, and dependencies (section 2.3.1) for each word in a sentence based on the spacy-NLP framework (which is used by the updated version of the TEMOS tool) [6].

For example:

```
sentence = [
    Token("Each", "DT", "det"),
    Token("confirmation", "NN", "compound"),
    Token("window", "NN", "nsubj"),
    Token("has", "AUX", None),
    Token("a", "DT", "det"),
    Token("close", "JJ", "amod"),
    Token("button", "NN", "dobj"),
    Token(".", ".", "punct")
]

    sentence[0].head = sentence[2]
    sentence[1].head = sentence[2]
    sentence[2].head = sentence[3]
    sentence[4].head = sentence[6]
    sentence[5].head = sentence[6]
    sentence[6].head = sentence[3]
    sentence[7].head = sentence[3]
```

The above collection represents the *pattern recognition*, and words can be connected (related) according to their head dependencies and parts of speech. For example, the item in position [0] receives information from the element in position [2]. Likewise, based on these dependencies, the relationships between each word should be created.

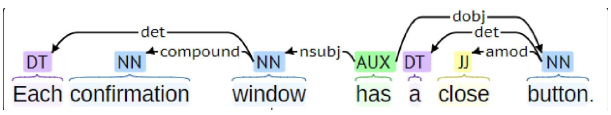

Figure 3.7: Pattern recognition

Subsequently, it should be stored and appear as a *pattern* in Neo4j as mentioned in Figure 3.7. Thus, the simplest possible way to do this is to store *words* as *nodes* with part of speech tags and dependency as *properties*. Meanwhile, relationships will be created as per *dependency head*. However, this manual approach is welcome when we are processing some sentence patterns. But, it is a strenuous task to be achieved when hundreds of sentence patterns want to be stored. On the other hand, it is not also a requirement and a goal to be obtained. So this has to be a dynamic approach that should work for any sentence with its grammar annotations. Hence, the feasible method is to iterate the elements of the sentence list and store the *word* attribute as *nodes*, and grammatical annotations as *properties* of the respective node. Following this, the header dependencies should be iterated to create a relationship between the nodes. Therefore, it sounds like it is accessible and iteration need not be burdensome to perform in Python. But then, how should these iterated values be saved in Neo4j?

Consequently, to make it viable *Py2neo* comes into play. Certainly, *py2neo* holds plenty of objects, functions, and constructors (section 3.2.0.3) to create nodes, relationships, and many more other Cypher query operations which are supported by Neo4j. Likewise, one among them is *Node object* which is used to create a node. [53] This constructor takes *label* and *properties* of the node in key-value pairs, as given below. On the other hand, I explained how the creation operation works with this constructor in section 3.2.0.3.

```
a = Node("Patient", name="John")
g.merge(a, "Patient", "name")
```

As a result, I devised the concept of providing iterated values to this *node constructor* from the collection of sentence patterns, which comprises *words, part of speech tagging, dependencies,* and *head dependencies*, i.e., the sender node for each node. Additionally, a link between those nodes can be established using either a direct cypher query for relationship or by the py2neo's relationship object, but the relationship should be generated on a periodic basis based on the head dependency. Secondly, this dynamic relationship construction is reminiscent of Neo4j's APOC procedure section 2.4.3.5, which includes a number of procedures, one of which appears to be useful for dynamically creating or merging relationships between nodes. Therefore, I can create relationships using either the **apoc.create.relationship** procedure [56] or the **apoc.merge.relationship** procedure [57], which avoids the development of duplicate relationships.

So far, I've gained comprehensive knowledge about how to retrieve a TEMOS sentence pattern list by iterating over it and storing it in Neo4j as a graph with nodes and relationships through the efficient *py2neo* interface. Next, extensive analysis is required to determine a feasible method for pattern

matching between the stored predefined linguistic patterns and the sentence patterns using a cypher query.

3.3.2 Finding a suitable way for pattern matching in Neo4j using a cypher query

This part will detail my examination of pattern matching between stored linguistic patterns as graphs in the Neo4j database, as well as how I came up with the concept of executing pattern matching using a Cypher query to accomplish the thesis's purpose. To begin with, let me explain the pattern matching that should be achieved.

I've given a visual picture below as an example:

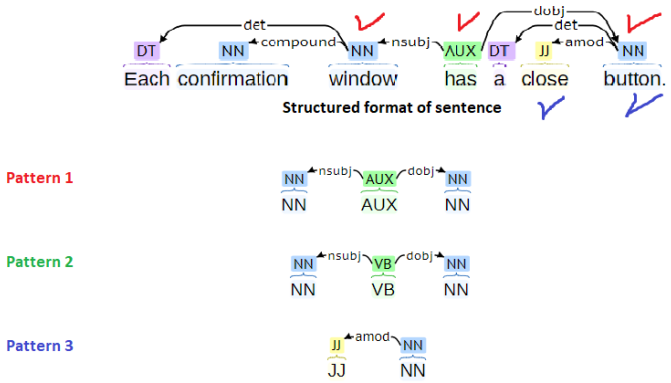

Figure 3.8: Example Sentence Pattern

The preceding image depicts a sentence pattern that has been matched with patterns 1 and 3, indicating that they are subgraphs of the sentence pattern due to the fact that those patterns are presented identically in the sentence pattern. For example, in pattern 1, the *AUX* node communicates the *nsubj* dependence connection to the *NN* node, while the same *AUX* communicates the *dobj* relationship to the other *NN* node. The same structure is revealed if we look for it in the sentence pattern. (I've highlighted those nodes in red.) Similarly, pattern 3 matched as well (I marked those nodes with blue to represent them).

Likewise, I'm required to design a Cypher query in the Neo4j database in order to discover common properties and relationships between the sentence pattern graph (which we can also refer to as a *Query* per requirement) and the other predefined linguistic graph patterns. This allows me to determine

which predefined patterns correspond to the sentence pattern graph (Query). So, what are the common properties and relationships that are referred to ?

Indeed, each node in the Query graph will have the words, part of speech tags, and dependencies as properties, and will be connected to the other nodes via the dependencies. Similarly, nodes of predefined linguistic patterns include characteristics indicating their parts of speech tagging and dependencies, and those nodes will be associated according to their dependence relationship. As a result, the properties and relationships are key in this case. Each node communicates with another by sending and receiving. Thus, by obtaining the POS property and kind of relationship that are conveyed from sender to receiver nodes of both graphs to be matched (say, Query and pattern 1), it is possible to determine whether the Query graph and pattern 1 share common POS characteristics and relationship types, and also whether they are in exact structure. If so, then I can deduce that both graphs are matched and one is a subgraph of the other.

For example, the following picture illustrates the common characteristics between the *Query* and *Pattern 1* graphs.

Query (Sentence) graph

Sender	Type	Receiver
NN	compound	NN
NN	det	DT
AUX	**nsubj**	**NN**
AUX	**dobj**	**NN**
NN	det	DT
NN	amod	JJ

Pattern 1

Sender	Type	Receiver
AUX	**nsubj**	**NN**
AUX	**dobj**	**NN**

Figure 3.9: Example to show equality between graphs

So far, I've come up with the idea of developing a Cypher query that performs statically. i.e., *Query graph* specifically checks with *patterns 1, 2, or 3*. However, I must construct my Cypher query to behave dynamically by comparing all *saved patterns* in the database to the *Query graph* and displaying the matched patterns as results. As a solution, the next subsection will present my analysis of dynamic execution.

3.3.2.1 Analysis and design of dynamic execution

Consequently, this need for dynamic operations drove me to investigate the possibility of Cypher query execution through an APOC procedure section 2.4.3.5,

as it contains several dynamic operations, such as for dynamically creating and merging nodes and relationships, including **apoc.create()** and **apoc.merge ()**. Thus, these methods sparked my curiosity about the possibility of a procedure or function that might dynamically run a whole Cypher. And indeed, I discovered a procedure called **apoc.cypher.run ()**, which is used to periodically perform Cypher operations. [58] Moreover, after discovering this resource, I had a vague notion that I might be able to execute my statically coded Cypher query in a dynamic way. But, how will this be possible?

Actually, each graph pattern will be saved in the Neo4j database with a unique label name such as *Query, pattern 1, pattern 2, pattern 3*, and so forth. Thus, I will retrieve all the labels from the database and then verify for matches to the *Query* graph by iterating through all the labels in the database and outputting those that are all matched. The following built-in procedure provided by Neo4j is useful for retrieving all labels in the database. It employs the CALL clause to invoke the procedure and the YIELD clause to hold the procedure's return values. [59]

```
CALL db.labels()
YIELD label
```

Following that, I'll incorporate my Cypher query into the **apoc.cypher.run()** procedure, which will iterate through all the returned labels and compare the graph patterns included under them to the Query graph for matching.

In summary, I've described my approach to dynamically matching patterns between the Query graph (sentence pattern) and all other stored patterns in the database in this part. As a result, I will employ this strategy in my implementation.

3.4 Summary

In overall summary, I conducted an analysis to determine the best method for creating an interface between Python and Neo4j in section 3.1

Then, under section 3.2, I determined that Py2neo would be the appropriate interface. Additionally, I explored its connection to Neo4j and database administration capabilities.

Finally, in section 3.3, I acquired a concept and a design for my Neo4j database, which will get a Query (sentence) from a TEMOS tool, do pattern matching, and output the matched patterns.

Chapter 4

Configuration Stage

Prior to entering into implementation, it is essential to provide specifics about the tools and software that will be required and how they will be deployed. So, I have added configuration details of each software separately.

4.1 Neo4j Database

Let's begin with the thesis's core software, the Neo4j database. Indeed, the Neo4j database is available in a variety of versions [60] and can be installed on Windows, macOS, or Linux. It is available as a console or desktop application. [61] However, the desktop application is useful for developers since it allows them to work with local Neo4j databases, manage multiple projects, and connect to remote Neo4j servers. Additionally, it includes a complimentary Developer License for the Enterprise Edition. [62]

Neo4j desktop software can be downloaded from the Neo4j Download Center [63], and it can also be updated using a graphical user interface. Additionally, it enables users to simply install extensions such as APOC and Graph Data Science as plugins. Moreover, once installed, it provides a platform for creating a local database within the desktop application due to the inclusion of Neo4j Enterprise Edition.

For the aforementioned reasons, I chose to use the Neo4j Desktop application, and I've provided the version I used below.

- **Neo4j Desktop 1.3.4.**
- **Neo4j 4.3.1 Enterprise Edition.**

4.1.1 APOC Installation

To take advantage of Neo4j's extra features and procedures, users should install the APOC (Awesome Procedures On Cypher) add-on library. This was

already discussed in the section 2.4.3.5. Additionally, this APOC library can be deployed as an extension plugin for the Neo4j Desktop application's local database. This is one of the Desktop application's advantages. Besides that, the APOC can also be installed manually by downloading the jar files and inserting them directly in the Neo4j/plugins folder. However, because the APOC relies on internal APIs, when manually installing Neo4j, it is necessary to use the appropriate version. For example, if Neo4j is version 4.1.0, the appropriate APOC version is 4.1.0.0. However, if APOC is installed as a plugin within the Neo4j Desktop, a suitable version is picked automatically based on the Neo4j database. [64]

Thus, in accordance with my Neo4j database, I used the **APOC version 4.3.0.0.**

4.2 Python setup

Python can be downloaded from their official website [65] and is available in different versions. I specifically used **Python 3.9.1** for my implementation.

4.2.1 Pycharm IDE

I chose to build my Python code using an IDE (Integrated Development Environment), which enables the analysis, execution, and debugging of code in a very sophisticated environment. As a result, I discovered *Pycharm*, a similar environment. It is intended for professional developers and is available in both a *professional* and a *community version*. [66] To build Python code, all I require is the community edition, which is completely free and open source. So, I installed it on my Windows operating system. I used the *community edition*, version **2021.1.2** in particular.

Additionally, there are various Python IDEs available. Pycharm was chosen primarily for my convenience and also for its most user-friendly graphical user interface (GUI). Meanwhile, the other tool, *Jupyter Notebook*, provides an excellent experience for writing Python in a *Web-based interaction* environment that is optimized for Python Notebooks, which are a hybrid of text documents and Python code. Interestingly, it is also feasible to interact with Neo4j from within a Jupyter notebook using Py2neo. And the Jupyter notebook can be downloaded from the official website. [67]

4.2.2 Py2neo driver installation

Using a community driver named Py2neo, I created a high-level API between Python and Neo4j. This driver can be installed using a pip installer, which is a Python package installer. So, after installing the pip package installer, we will

be able to install any package from the Python package index. [68] Likewise, Py2neo can be installed through the console using the following command:

pip install py2neo

When it comes to versioning Py2neo, it uses the YYYY.N.M scheme since the year 2020. N is the yearly incrementing zero-based number, and M is a modification inside that version. Additionally, the following Python and Neo4j versions are supported for Py2neo installation: [69] [70]

- Python versions 2.7/3.4/3.5 (and above)
- Neo4j versions 3.4/3.5/4.0 (and above)

I specifically utilized **Py2neo version 2021.2.3** for my implementation.

4.2.3 Pytest

A test should be run to confirm that the code operates well and generates accurate results. As a result, I'll be testing Python programs using a framework called *Pytest*. [71] This can be installed using the below command from the Python Package Index (PyPI). [72]

pip install pytest

I installed **Pytest version 6.2.4** particularly to create tests for my code implementation.

4.3 Summary

In summary, I've installed all the essential tools and software. Meanwhile, I described the version and method of installation that I used. Finally, all of the elements are in place and the process of executing this thesis implementation can begin.

Chapter 5

Implementation

I'm going to work on the implementation process in this chapter using the resources gathered and concepts developed in the chapter 3. First, I will thoroughly define the procedure and replete it with a flowchart design for clear representation. Following that, I'll commence carrying out each and every strategy necessary to accomplish this thesis objective. Moreover, I will write some test cases to verify that the implemented code performs well and meets the criteria, as well as to address any unsatisfied conditions through implementation modifications. Finally, I will finish this chapter with a brief overview of the accomplishments.

5.1 Design for Implementation

This section covers the implementation process's flowchart, which is illustrated in Figure 5.1. And I offered a brief description of each subsequent step.

1. To begin, it stores specified patterns in the Neo4j database, either directly using the Neo4j graphical user interface (GUI) with the use of Cypher queries or through the use of the Py2neo interface and associated Cypher functions.

2. The following step is to accept a sentence pattern (Query) for analysis as a list using a shared interface named *get_matched_patterns* from a TEMOS tool. Following that, iteration of the received list will occur in order to access the *words, part of speech tags, and dependencies*, before converting them to graphs using Py2neo functions and Cypher queries and storing them in a Neo4j database.

3. Then, the predefined patterns are matched to the sentence pattern (Query), and the results are reported.

4. Finally, the sentence pattern graph (Query) will be erased from the database to create room for the analysis of another new sentence pattern, and procedure 2 will proceed.

5.2 Storing the predefined linguistic patterns in a Neo4j Database

After successfully deploying the Neo4j database as per the section 4.1, I began my experiment by storing the predefined linguistic patterns from the research papers [2], [3], [4], and [5]. However, before we begin, how are the patterns to be stored in Neo4j?

Indeed, this is achievable in two ways:

- Cypher queries can be used to store items directly in Neo4j's graphical user interface (GUI).
- Additionally, another way is to include Cypher queries straight into Python code, and then save the patterns in Neo4j using Py2neo subgraph operations, as defined in section 3.2.0.3.

Thus, I will demonstrate how I implemented both of the aforementioned ways for storing specified patterns in Neo4j.

5.2.1 Direct way of storing patterns using Neo4j's GUI

If we use this method, we must manually write a Cypher query to create nodes and relationships based on the patterns using the Cypher CREATE keyword. As with the basic experiment of creation that I described previously in the section 2.4.3.3, I will use a similar approach for storing linguistic patterns here.

I began my experiment by randomly selecting one of the defined linguistic patterns named *Attribute Pattern 1* from section 2.3.1.1 and constructing a cypher query as given below to store in Neo4j based on pattern rules.

```
CREATE
 (n1:AttributePattern1 {word: "NN", POS: "NN"}),
 (n2:AttributePattern1 { word: "VB", POS: "VB"}),
 (n3:AttributePattern1 {word: "NN", POS: "NN"}),
  (n2)-[:nsubj]->(n1), (n2)-[:dobj]->(n3);
```

As a result, Neo4j now contains the pattern. And Figure 5.2 depicts the corresponding screenshot.

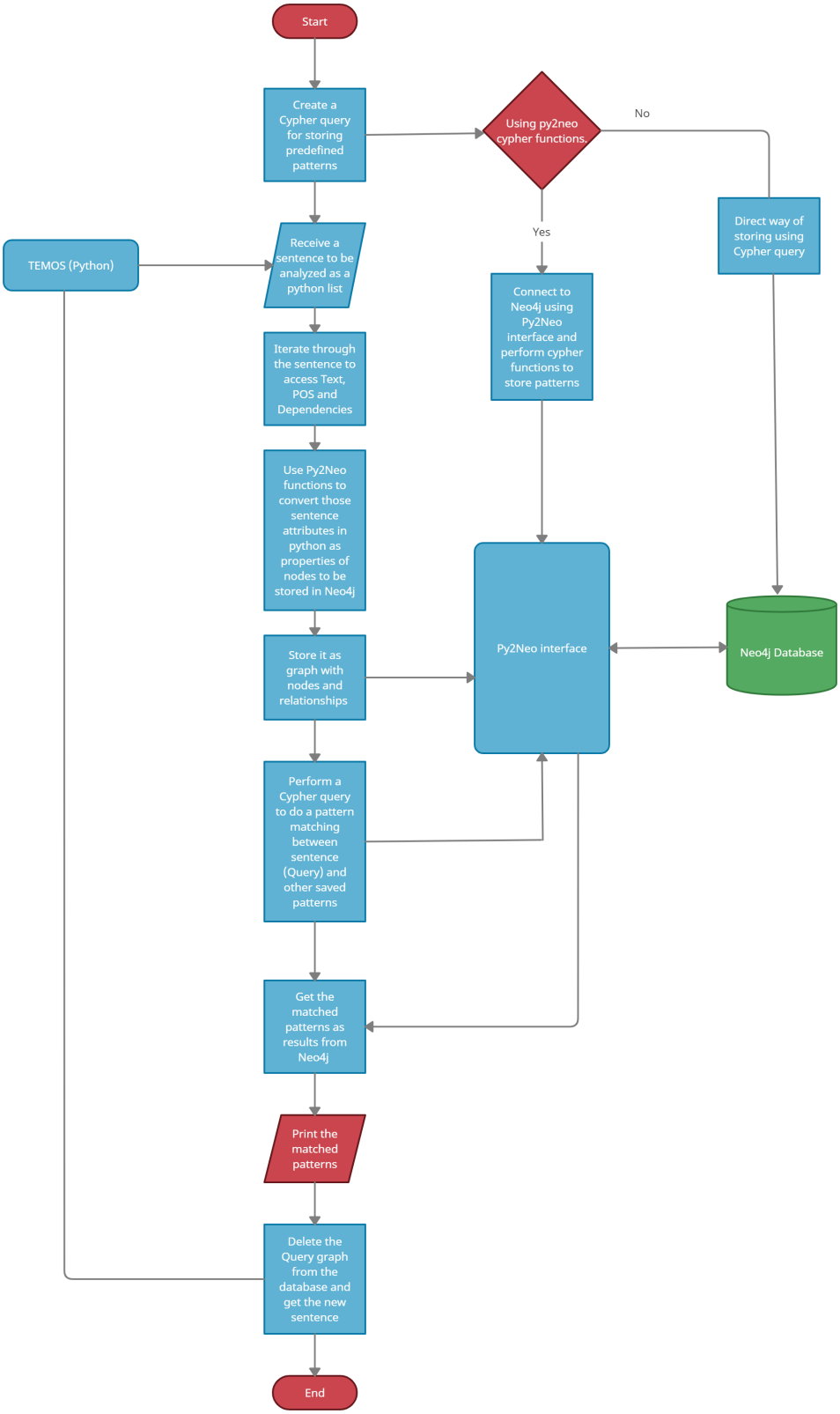

Figure 5.1: Flowchart design for implementation

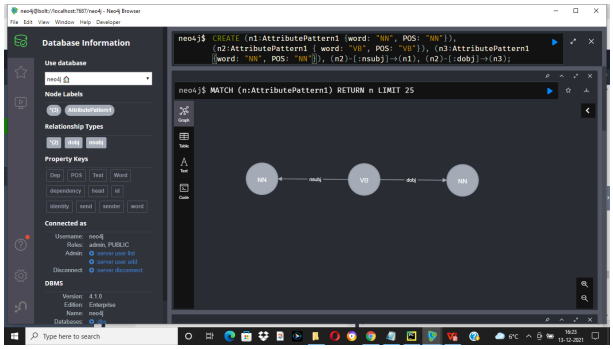

Figure 5.2: Screenshot for pattern storage using Neo4j GUI

Similarly, the other predefined patterns can be saved in Neo4j by explicitly configuring a cypher query in the Neo4j browser (GUI). Additionally, instead of utilizing a CREATE clause in a cypher, we may use a MERGE clause. Because MERGE is also a writing clause for the cypher and a mixture of CREATE and MATCH. Indeed, the MERGE clause's advantage is that it prevents the construction of the same graph twice by determining if the identical nodes and relationships already exist in the database. Otherwise, it generates the graph; if not, it skips the construction. A MERGE query is constructed similarly to a CREATE query, except that the CREATE clause is replaced by a MERGE clause. As a result, in order to avoid duplicating pattern storage, I chose to utilize the MERGE procedure to store further patterns.

Moreover, after dealing with the direct method of saving predefined patterns in Neo4j, I choose to utilize a different strategy for storing additional patterns, i.e storing them using Python code that incorporates Py2neo cypher methods. Because I will be interacting with the TEMOS tool using Python to obtain a sentence pattern (Query) for pattern matching. Thus, it would be more efficient if my Python code had distinct methods for inserting (storing) specified patterns into the database and retrieving sentence patterns from the TEMOS tool for analysis. As a result, in the next section, I will apply and describe this strategy.

5.2.2 Storing patterns using Python code

Prior to implementing this strategy, the code must be designed with the relevant classes, attributes, and functions. Thus, I imported the Graph and Node objects from Py2neo modules into the main code (which is named graph_patterns_checker), whereas the Graph object is used to establish a connection to Neo4j and create an interface for working with Neo4j from Python (as previously explained in the *Database management* part of section 3.2.0.1). [49] Additionally, the Node object is utilized to create the node

in Neo4j database. [53]

```
from py2neo import Graph, Node
```

Following that, I constructed a class entitled *Token* that contains an internal initializer function for initializing the class's attributes, including *text, pos, dep, and head*. These attributes are utilized to get access to the linguistic pattern's properties. Additionally, the *self* parameter is used to associate the current instance of the class, and also providing access to the class's attributes.

Below I have added the class diagram of Token class with its methods and fields for representation.

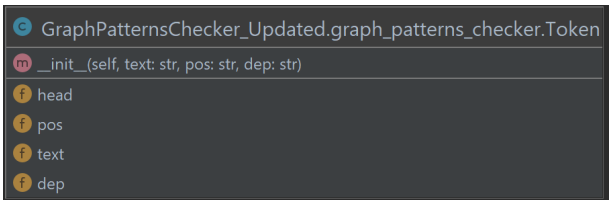

Figure 5.3: Class diagram for Token class

On the other hand, I created a new class called *GraphPatternsChecker* that begins with an initializer function that establishes a connection to Neo4j by passing the connection's individual settings, such as *uri, username, and password*, to the Graph () constructor in accordance with Py2neo's connection profile. Following that, I created two functions. One is to save the predefined patterns in Neo4j, and the other is to do pattern matching on the TEMOS-retrieved sentence using a Cypher query and eventually outputting the matched patterns as results.

Now let's focus on the function insert_patterns which is to store predefined patterns in Neo4j. However, before going into the detail about this function behaviour. It is necessary to give limelight to the other class called *Temos-GraphInitializer*. So, what is this class?

Actually, this is the class that initializes the insert_patterns method, which performs the storing process. Additionally, this class is produced in a distinct Python file named temos_graph_initializer and imports the classes GraphPatternsChecker and Token. Thus, the primary purpose of this class is to have an initialize () function that generates a list of objects by adding *Token* class instances to a list, and each index of the list points to and accesses the *Token* class's instance attributes and methods.

For example, I generated a list pattern 1 as shown below, where each collection links to an instance attribute of the class *Token* via a value.

```
pattern_1 = [
            Token("NN", "NN", "nsubj"),
            Token("AUX", "AUX", None),
            Token("NN", "NN", "dobj")
        ]

pattern_1[0].head = pattern_1[1]
pattern_1[2].head = pattern_1[1]
```

When the first element in a list is considered, it really sends the values to each attribute, such as,
$NN \rightarrow text$
$NN \rightarrow pos$
$nsubj \rightarrow dep$

Similarly, other collections follow a similar pattern. Then, I assigned the head dependencies by accessing the collection at a specified index corresponding to a dependency on another collection.

Consider the following assignment, which indicates that the head attribute of the element at index 0 will be the element at index 1. Thus, pattern _1[1] communicates a connection to pattern _1[0], and the two are mutually reliant. Similarly, the remaining head dependencies can be fulfilled.

```
pattern_1[0].head = pattern_1[1]
```

In general, this method of assigning values to Token class instance attributes (text, pos, dep, and head) resembles the way nodes and relationships are created in a Cypher query. However, how?

Indeed, if we consider the following cypher query, it constructs each node together with its associated properties (text, pos, and dep) and relationships between the nodes, using dep as the relationship type.

```
CREATE
  (n1:Pattern1 {text: "NN", pos: "NN", dep: "nsubj"}),
  (n2:Pattern1 {text: "AUX", pos: "AUX"}),
  (n3:Pattern1 {text: "NN", pos: "NN", dep: "dobj"}),
  (n2)-[:nsubj]->(n1), (n2)-[:dobj]->(n3);
```

As a result, it demonstrates that each index element in the list includes the properties of each node, while the head dependencies indicate the nodes' relationship.

Now that this list of objects has been created, how will it be sent to the method insert_patterns for storage in Neo4j?

Indeed, this is achievable by creating an object for the class *GraphPatternsChecker* that has the function insert_patterns, and then calling that function with the constructed list as an argument. Thus, in the following example, graph_patterns_checker is the created object, and I called the function insert_patterns by passing the string *pattern1* to say, under this label name, store it in the database, and pass the list pattern_1 which contains information about the nodes, properties, and relationships should be created.

```
graph_patterns_checker = GraphPatternsChecker()
graph_patterns_checker.insert_patterns("pattern1",
pattern_1)
```

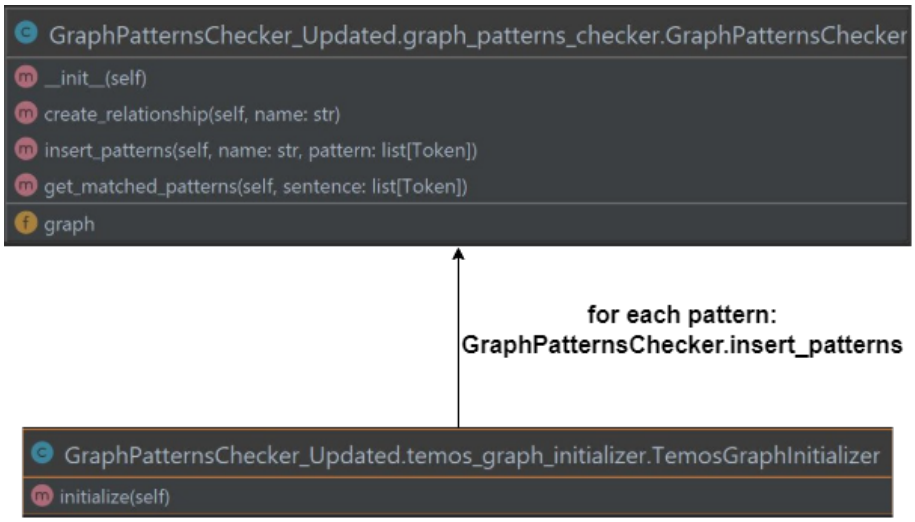

Figure 5.4: Class diagram for GraphPatternsChecker and TemosGraphInitializer

5.2.3 How the insert_patterns function behaves?

On the other side, let's see how the function insert_patterns operates and conducts the storing operation in Neo4j when it is called. To illustrate the function clearly, I have included a flowchart in Figure 5.5.

Before we begin, let's have a look at the insert_patterns function's parameters.

```
def insert_patterns(self, name: str, pattern: list[Token])
```

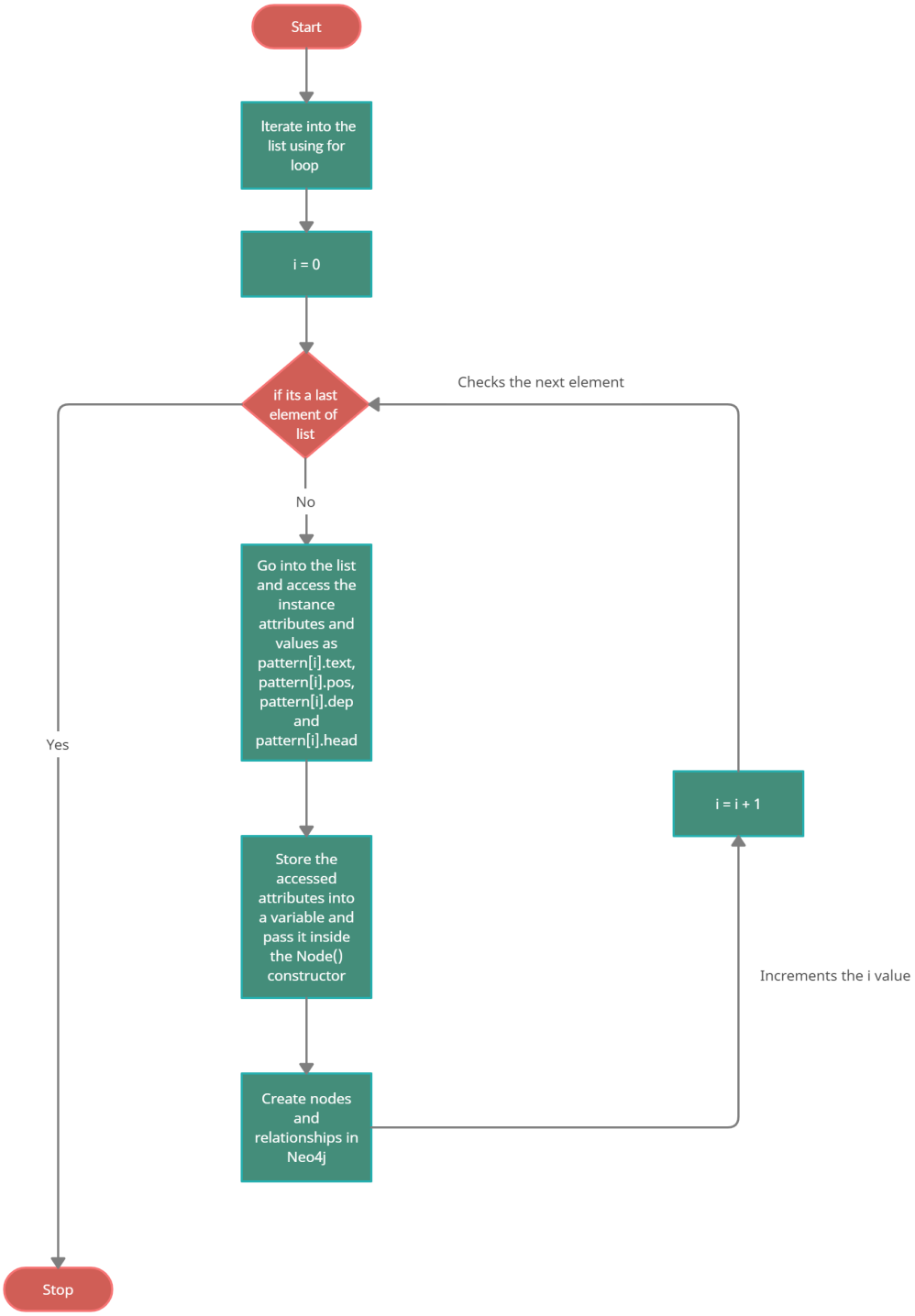

Figure 5.5: Flowchart for Storing operation

In fact, it is constructed using the following parameters.

- **self** parameter is used to get the current instance of the class. For example, the Graph() constructor which initiated the connection in the __init__can get accessed inside the insert _patterns function by using a self parameter.
- **name(str)** is to get the name of the pattern which passed as string.
- **pattern(list[Token])** get the passed list which holds the Token class's instances in it.

Now for the insert_patterns function's behavior. To begin, I iterated over the list passed by the *TemosGraphInitializer* class as an input.

To do this, I utilized a for loop in Python combined with the *range ()* function, which iterates through the list a specified number of times. Indeed, the range () function begins at 0 and iterates up to the value specified within. Subsequently, I used the len() method to determine the list's total length (number of elements) and handed it to the range function as range (len(pattern)). As a result, it iterates over the list's elements until it reaches the final one. For instance, if the list has three elements, the length will be three, and iteration will continue until the third element is reached.

Following that, once I'm within the list, I can access the instance attributes included inside it by referencing the index, such as pattern[i].text, pattern[i].pos, and pattern[i].dep. However, when it comes to the head values, it is transmitted as,

```
pattern_1[0].head = pattern_1[1]
pattern_1[2].head = pattern_1[1]
```

If I access the 0th index's head value in a similar way as done for other attributes, such as pattern[i].head, I will obtain the given value pattern_1[1]. However, it is impossible to store this information as a node property and it is also difficult to create dependent relationships. As a result, while looking for a solution, I came up with the idea of utilizing a Python built-in function called *list index*, which returns the index of the supplied element in a list. As a consequence, I provided the pattern[i].head attribute to the index method, and obtained the index value of the element containing the head attribute.

For instance, if pattern[0].head = pattern[1]. However, rather than returning the actual value of pattern[1], I will obtain the element's index, which is 1. After that, this integer value can be provided as the node's head property. Similarly, this can be done for other dependencies.

Additionally, when accessing each head attribute in a list, if any element in the list lacks a head-dependent value, Python may throw a NoneType

Attribute Error. For example, if the 0th and 2nd position items have head dependence but the 1st position element is None, the NoneType error will occur. As a result, I utilized an if-else condition to determine if the pattern[i].head is empty or not. If it is not none, the index element of the pattern[i].head is retrieved and placed in the variable named sendernode, where it will be utilized as a node property. However, if pattern[i].head is null, that location is skipped and does not result in the NoneType Attribute Error.

Thenceforth, I supplied all the accessed instance attributes from the list as node properties within the Py2neo's Node () constructor, as shown below. And the value of the iteration, which should begin at 0, should be saved in that *id* field. Additionally, the name property specifies the label name for which the nodes can be stored, and its value is passed as a string.

```
Node(name, id=i, POS=pattern[i].pos, Word=pattern[i].text,
Dep=pattern[i].dep, head=sender_node)
```

Then, this Node() constructor is assigned to a variable named node as a subgraph. The subgraph is then used as a constructor within the Py2neo's graph. merge() function. (As stated previously in section 3.2.0.3) Additionally, the label name (primarykey) and id (propertykey) will be included, so that the merging process can determine whether or not this node already exists in the database using these keys.

```
self.graph.merge(node, name, "id")
```

As a result, this merge process will create nodes in Neo4j from the iterated attributes from a list. However, how will the relationships between those nodes be established? Indeed, this is possible through the development of a Cypher query that establishes relationships between nodes based on their head dependencies. However, this brings up the question of under what conditions is this even possible?

Evidently, in response to this topic, I came up with a solution that connects the nodes based on their *id* and *head* properties. By and large, the created nodes have the properties shown in Figure 5.6. As we can see from the figure, the head value of the first node (table) is 1, which indicates that it should rely on and obtain a connection with the node with the id = 1, which meets the criteria for the second node (table). As a result, both nodes should be connected as shown in Figure 5.7. Thus, by comparing the *id* value of one node to the *head* value of another node, relationships may be created, and the *dep* property value of the recipient node serves as the *relationship type*.

Subsequently, I constructed the query shown below, which connects one node to another when the conditions outlined above are fulfilled. Thus, this query begins with the MATCH keyword, which considers two distinct nodes stored under the same label name pattern1, and I used variables p and q as

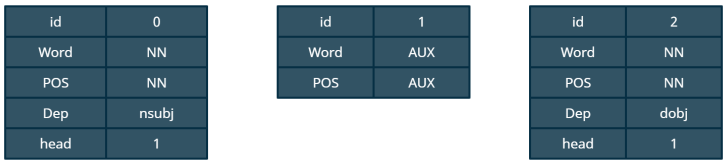

Figure 5.6: Properties of Nodes represented as table

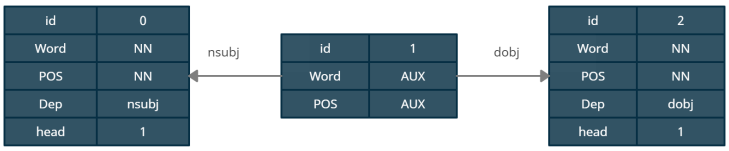

Figure 5.7: Properties of Nodes along with relationship

node references, gaining access to node properties such as p.id (for Node 1) and s.head (for Node 2), and checking for equality between those properties using WHERE conditions. Then, in a cypher query, I used the WITH term. Additionally, this term is utilized to introduce the variables p and q, which we already used within the WHERE condition. Indeed, if we utilized the WHERE condition, the results should have been returned immediately. However, because I need to conduct more operations, I send the condition results to the WITH keyword by creating variables p and q. Generally, the WITH keyword is used to change the output prior to it being given. As a result of this procedure, I'm required to establish relationships between the nodes that satisfy the equality constraint. Moreover, it should be generated dynamically for each node. In this regard, I created an APOC method (section 2.4.3.5) that conducts the relationship merging process. This procedure accepts the following parameters as input: [73]

- startNode
- relationshipType
- identProps
- onCreateProps
- endNode
- onMatchProps

In our situation, we just require the startNode, relationshipType, and endNode parameters to construct relationships. The rest of the arguments would be null. As a consequence, I set the sender node's variable p to the startNode, the receiver node's dependency property value to the relationshipType within the toString () function, because the relationshipType parameter of APOC merge supports STRING, and the receiver node's variable to the endNode. Following that, the procedure will execute and return the formed relationships using the YIELD keyword and an output parameter named rel. Finally, the RETURN keyword returns the relationships that were built in compliance with the equality constraints utilizing the APOC function.

```
MATCH(p:pattern1),(s:pattern1)
WHERE p.id = s.head
WITH p, s
CALL apoc.merge.relationship(p, (toString(s.Dep)), {}, {}, s, {})
YIELD rel
RETURN p, s
```

As a result, by directly performing the above Cypher query in Neo4j's GUI, the relationships between nodes are established perfectly. To run this query from Python, however, I must include it in Py2neo's run () method, as shown below.

```
self.graph.run("MATCH(p:" + name + "),(s:" + name + ")"
               "WHERE p.id = s.head WITH p, s "
               "CALL apoc.merge.relationship(p, "
               "(toString(s.Dep)), {}, {}, s, {}) "
               "YIELD rel RETURN p, s")
```

Following that, I encapsulated this Cypher run operation within a distinct function named create_relationship (self, name: str). Correspondingly, this function is called from the insert_patterns function with a *label* name as an argument, and in response, the create relationship function obtains a *label* name and passes it to the MATCH keyword within the query, where it performs the relationship operation between the nodes stored under that desired label name.

In summary, I've programmed the insert_patterns function to iterate over a list in order to access the instance attributes sent by the *TemosGraphInitializer* class, and then to use those attributes to store the predefined patterns in Neo4j as a graph with nodes and relationships using the Cypher query and Py2neo functions.

```
for i in range(len(pattern)):

    if pattern[i].head is not None:
```

```
                sender_node = pattern.index(pattern[i].head)

    else:
                sender_node = None

    node = Node(name, id=i, POS=pattern[i].pos,
                      Word=pattern[i].text,
                      Dep=pattern[i].dep,
                      head=sender_node)

    self.graph.merge(node, name, "id")
    self.create_relationship(name)
```

As a result of this approach, the graph formed in Neo4j will look as seen in Figure 5.8.

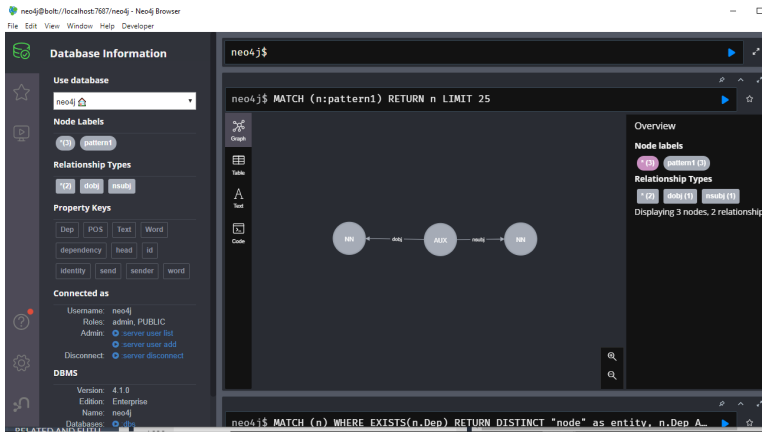

Figure 5.8: Pattern stored as graph

Similarly, I built the other predefined patterns as a list of objects in the initialize () function of the TemosGraphInitializer class, sent them to the insert pattern function, and stored them as graphs in Neo4j (Figure 5.9 illustrates this with a screenshot). Additionally, this technique enables the addition of any other patterns in the future.

After saving the predetermined patterns, it's necessary to write code that retrieves the sentence pattern from the TEMOS tool and conducts the pattern matching procedure. Let's have a look at this in the next part.

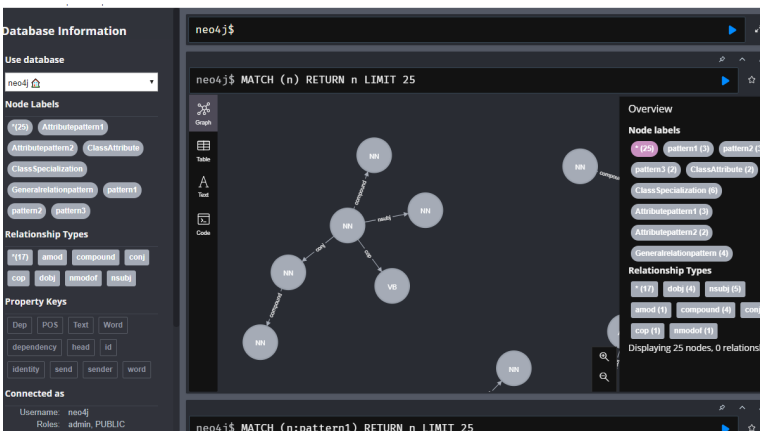

Figure 5.9: Patterns stored as graph

5.3 Getting the Sentence as an Input to the Neo4j Database and perform pattern matching

Now we'll look at the process of obtaining sentence patterns from a TEMOS tool and storing them in Neo4j for pattern matching and result generation. To do this, I developed a new function named,

```
get_matched_patterns(self, sentence: list[Token])->list[str]
```

Moreover, this function accepts arguments such as:

- **self** parameter is used to get the current instance of the class. For example, the Graph() constructor which initiated the connection in the __init __can get accessed inside the get_matched_patterns function by using a self parameter.
- **sentence: (list[Token])** get the passed list which holds the Token class's instances in it. Additionally, the function will generate a sentence graph in Neo4j by accessing these attributes.

Secondly, the function's return type is a list of strings, marked as list[str].

On the other hand, when it comes to behavior, this method behaves similarly to the other function named *insert_patterns* that we have seen in section 5.2.3. The distinction is that we are now obtaining a list of items via the TEMOS tool. Thus, the procedure of iterating over the list, accessing instance properties, and giving them to the Py2neo Node () constructor for the purpose of creating nodes and relationships is identical to what I did in the insert_patterns method. Additionally, the minor adjustments is to the node's label name, which would be *Query*. Because it is not possible to provide a

unique label name to nearly hundreds of words transmitted by TEMOS. Thus, each sentence received will be saved in Neo4j for pattern matching analysis under the fixed label name *Query*, and after the pattern matching is complete, the *Query* graph will be erased and a new sentence will be received for analysis under the label *Query*, and so on.

```
Node("Query", id=i, POS=pattern[i].pos, Word=pattern[i].text,
Dep=pattern[i].dep, head=sender_node)
```

Furthermore, *get_matched_patterns* is accessed by constructing an object for the class *GraphPatternsChecker* that has the method get_matched_patterns, and then passing the sentence (list) as an argument to a function call performed using the object. For instance,

```
graph_patterns_checker = GraphPatternsChecker()
graph_patterns_checker.get_matched_patterns(sentence)
```

After receiving the sentence to be saved and analyzed in Neo4j, the first get_matched_patterns method will complete the storage process by iterating over it, accessing the attributes, and building nodes and relationships using the Cypher and Py2neo functions. Moreover, the create_relationship function is called with a fixed label name, *Query*, and in response, relationships between the nodes stored under that label name are created.

```
for k in range(len(sentence)):

    if sentence[k].head is not None:
       head_value = sentence.index(sentence[k].head)

    else:
       head_value = None

    node = Node("Query", id=i, POS=pattern[i].pos,
                        Word=pattern[i].text,
                        Dep=pattern[i].dep,
                        head=sender_node)

    self.graph.merge(node, "Query", "id")
    self.create_relationship("Query")
```

Thus far, we've looked at the process of retrieving and saving sentences from the TEMOS tool in Neo4j. Following that, it's time to perform pattern matching on the sentence and the predefined patterns, resulting in the generation of the matched ones.

5.3.1 Matching the Query/Sentence graph with the predefined patterns

In this part, I will develop a Cypher query to perform pattern matching between the Query (sentence) graph and predefined patterns based on the resources and thoughts gathered throughout the pattern matching analysis section (section 3.3.2).

To begin, I write a Cypher query that performs pattern matching between a stored predefined pattern and the query graph *statically*. For this experiment, I utilized the sample sentence pattern list with instance attributes (also mentioned in section 3.3.1) in accordance with the criteria, and I saved it in the Neo4j database under the label *Query*, as seen in Figure 5.9.

Following the database storage of the *Query* and predefined pattern graphs, the time has come to develop a Cypher query for pattern matching.

Subsequently, I developed a Cypher query that does pattern matching and returns *true* if the two patterns match. That is, a graph is a subgraph of another graph. I performed this Cypher query using the *Query* graph and the predefined pattern contained in the database (a dummy pattern called *pattern1* created for experimentation).

```
MATCH (a:pattern1)-[r1]->(b)
WITH collect({SenderPOS:a.POS,relation:type(r1),
              ReceiverPOS:b.POS}) AS pattern
 CALL
 {
  MATCH (n:Query)-[r]->(p)
  RETURN collect({SenderPOS:n.POS,relation:type(r),
  ReceiverPOS:p.POS}) AS query

 }

 WITH pattern as p,query as q
 WHERE ALL (a IN p WHERE a IN q)
 WITH apoc.coll.duplicatesWithCount(p) AS dup_pattern,
 apoc.coll.duplicatesWithCount(q) AS dup_query
 WHERE ALL (a IN dup_pattern WHERE a IN dup_query)
 RETURN 1
```

I've described the behavior of the designed Cypher query in detail below:

1. To initiate, I'm checking the outgoing relationship of each node in the *pattern1* graph using the MATCH keyword. And, as previously explained in section 3.3.2, each node in the graph is connected to another via sending or receiving; thus, by obtaining the part of speech (POS)

property of the sender and receiver nodes, as well as the type of relationship communicated, I am able to compare these characteristics for matching to those of the other graph, which were also obtained in a similar manner.

2. Then, using Cypher query's aggregating function named *collect()* under the WITH keyword [74], I aggregate the *POS* property (sender and recipient nodes) and the communicated *relationship type* into a collection called *pattern*. Because the WITH keyword is used, the collection may be used in subsequent query operations. Additionally, this is one of the primary benefits of the WITH keyword. [75]

3. Following that, a collection mechanism akin to the *pattern1* graph has been implemented for the *Query* graph. However, this process is performed as a subquery within the CALL clause. The rationale for creating it as a subquery is to aggregate the properties of the *Query* graph independently of the collection of *pattern1*. The *collect()* method is used inside the RETURN keyword in this subquery, and the collected data is saved in a collection named *query*. Then, outside of the subquery, this collection can be utilized for other actions or returned.

4. Following that, I'm manipulating both the *pattern1* and *Query* graph collections using the WITH keyword rather than returning since I have other operations to complete. As a result, I've added a variable p to represent a collection named *pattern* and a variable q to represent a collection named *query*.

5. To determine if the collections have common properties, I utilized Neo4j's predicate function *all ()*. Indeed, this *all()* method determines whether or not the elements in a list are present in another list, which is actually employed in conjunction with the WHERE condition constraint. [76] Thus, by using the WHERE ALL condition, I'm determining whether or not elements in the *pattern (p)* collection exist in the *query (q)* collection. Moreover, a is the variable's representation of an element.

6. After executing the equality condition, I persisted on verifying the equality of duplicate elements that exist in both collections. Thus, what would this duplicate element be and what impact will it have?

   Indeed, as seen in Figure 5.10, a relationship between *NN* and *JJ* with the relationship type *amod* occurred twice. Therefore, if that is the case, the other pattern with which it is matched should also include the same type of relationship in the same number of counts. Otherwise, the incorrect outcomes will be obtained. Therefore, I utilized an APOC procedure named *apoc.coll.duplicatesWithCount()* which looks for duplicate items and their counts in a list or collection and provides them in

JSON format. [77] Thus, I used collection *pattern (p)* to do a check and recorded the duplicate items and their count in a collection named *dup_pattern*. This was also done for collection *query (q)*, which was saved as a *dup_query*. Following that, I used the WHERE ALL condition to determine whether or not the duplicate element of the pattern and the query were identical.

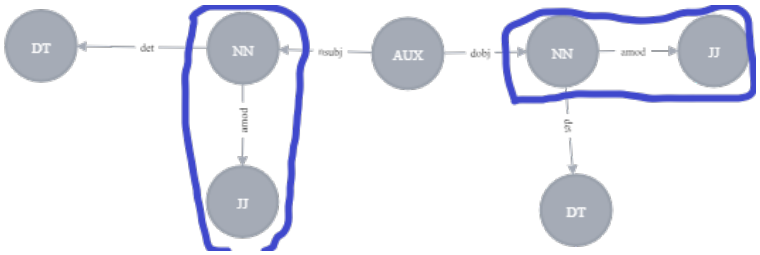

Figure 5.10: Sample pattern to represent duplicates

7. Finally, if all requirements are met, *pattern1* is a subgraph of *Query* and is matched. Thus, using the RETURN keyword, *1* will be printed as a result.

As a consequence, when I execute this Cypher query straight from Neo4j's GUI, I obtain the value *1* since *pattern1* matches the *Query*. Figure 5.12 includes a screenshot of the execution.

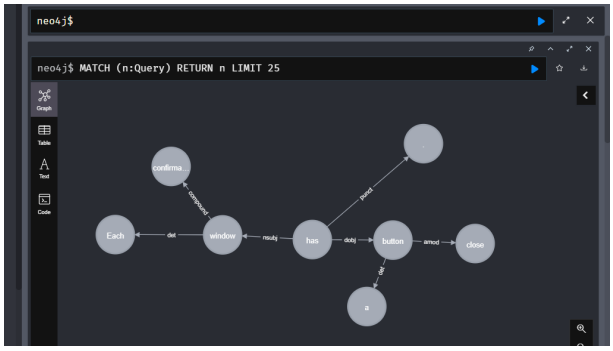

Figure 5.11: Query graph

To demonstrate how they are related and how *pattern 1* is a subgraph of *Query*, I returned collection lists including the elements *pattern (p), query (q), dup_pattern,* and *dup_query*, as seen in Figure 5.13. It is represented in the form of JSON elements.

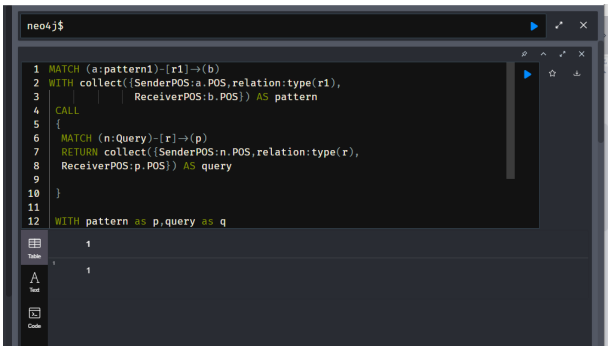

Figure 5.12: Screenshot for Result

Figure 5.13: Screenshot of printing collections

Additionally, to facilitate visualization, I have displayed it as a table, as seen in Figure 5.14. The highlighted items are those that match. Furthermore, *pattern (dup_pattern)* has no duplicate components, but *query (dup_query)* contains them. However, it doesn't matter if the *Query* graph has duplicates or more components in comparison to the *pattern*. Because we're determining whether or not the *pattern* is a subgraph of *Query*. Thus, if all items in the *pattern*, including duplicates (if any), are presented in *Query*, then only the *pattern* is matched and a subgraph of *Query*.

For instance, I compared the graph of another sample pattern to Query, which also has duplicate elements, as seen in Figure 5.15. As a consequence, it is matched against Query, including duplicate items, and I have re-added the collected JSON elements in a tabular style for clarity. (See Figure. 5.16.)

Likewise, I experimented with the constructed Cypher query by comparing it to the other patterns; if they satisfy all of the requirements, result 1 is printed. For instance, a sample pattern named *pattern 2* is not matched with Query, and while *pattern 5* has duplicate components identical to those in *Query*, it fails to match with the remaining elements, resulting in the pattern being unmatched with the Query graph. I've illustrated the patterns below

p	q	dup_pattern	dup_query
{ "SenderPOS": "AUX", "ReceiverPOS": "NN", "relation": "dobj" }	{ "SenderPOS": "NN", "ReceiverPOS": "NN", "relation": "compound" }		{ "count": 2, "item": { "SenderPOS": "NN", "ReceiverPOS": "DT", "relation": "det" } }
{ "SenderPOS": "AUX", "ReceiverPOS": "NN", "relation": "nsubj" }	{ "SenderPOS": "NN", "ReceiverPOS": "DT", "relation": "det" }		
	{ "SenderPOS": "AUX", "ReceiverPOS": ".", "relation": "punct" }		
	{ "SenderPOS": "AUX", "ReceiverPOS": "NN", "relation": "dobj" }		
	{ "SenderPOS": "AUX", "ReceiverPOS": "NN", "relation": "nsubj" }		
	{ "SenderPOS": "NN", "ReceiverPOS": "JJ", "relation": "amod" }		
	{ "SenderPOS": "NN", "ReceiverPOS": "DT", "relation": "det" }		

Figure 5.14: Collections of pattern1 and Query in a Table format

Figure 5.15: Sample pattern

by emphasizing the presented and unpresented parts in comparison to *Query*. Additionally, I've re-added the Query graph image (Figure 5.18), but this time with the nodes captioned with their POS properties to facilitate comparison.

Moreover, I developed some additional example patterns and verified them using the Cypher query. As a consequence, it produces flawless outcomes. Prior to finalizing the solution, it is vital to evaluate the intended query by creating test cases and examining the risk factors to analyze to what extent it works properly. As a result, I have written a test case in the following section.

5.3.2 Test Cases

To determine whether the query works correctly in all cases. I documented several test cases and verified each one (as illustrated in Figure 5.20). I utilized an online test planning application called TESTPAD for this purpose. [78]

p	q	dup_pattern	dup_query
{ "SenderPOS": "NN", "ReceiverPOS": "DT", "relation": "det" }	{ "SenderPOS": "NN", "ReceiverPOS": "NN", "relation": "compound" }	{ "count": 2, "item": { "SenderPOS": "NN", "ReceiverPOS": "DT", "relation": "det" } }	{ "count": 2, "item": { "SenderPOS": "NN", "ReceiverPOS": "DT", "relation": "det" } }
{ "SenderPOS": "AUX", "ReceiverPOS": "NN", "relation": "dobj" }	{ "SenderPOS": "NN", "ReceiverPOS": "DT", "relation": "det" }		
{ "SenderPOS": "AUX", "ReceiverPOS": "NN", "relation": "nsubj" }	{ "SenderPOS": "AUX", "ReceiverPOS": ".", "relation": "punct" }		
{ "SenderPOS": "NN", "ReceiverPOS": "DT", "relation": "det" }	{ "SenderPOS": "AUX", "ReceiverPOS": "NN", "relation": "dobj" }		
	{ "SenderPOS": "AUX", "ReceiverPOS": "NN", "relation": "nsubj" }		
	{ "SenderPOS": "NN", "ReceiverPOS": "JJ", "relation": "amod" }		
	{ "SenderPOS": "NN", "ReceiverPOS": "DT", "relation": "det" }		

Figure 5.16: JSON elements of sample pattern and Query

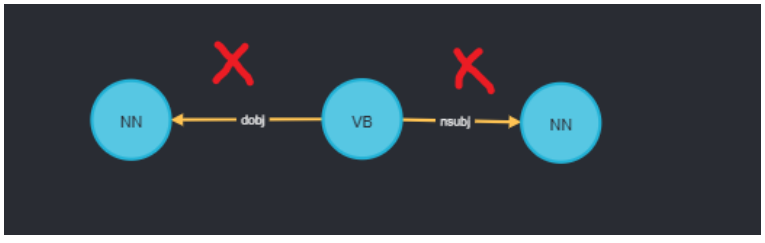

Figure 5.17: Pattern 2

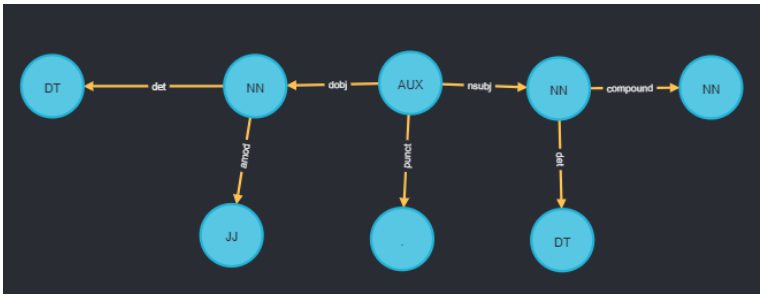

Figure 5.18: Query graph with POS caption for nodes

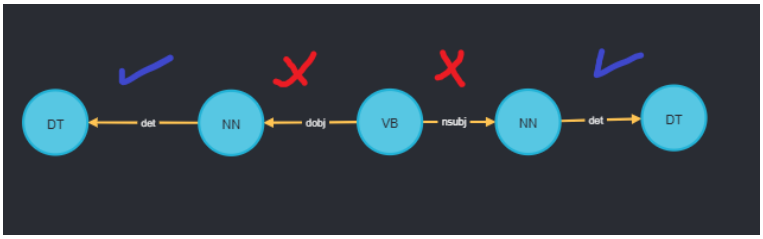

Figure 5.19: Pattern 5

0001	**Check the outgoing relationship and collect the POS and relationship type of the sender and receiver nodes**	
0002	Examine the stored pattern's outgoing relationships and collect the POS (Sender & Receiver) and relationship type, then store them in a list named "pattern"	✔
0003	Examine the Query's outgoing relationships and collect the POS (Sender & Receiver) and relationship type, then store them in a list named "query"	✔
0004	Display both lists and check that they are appropriately stored	✔
0005	**Ascertain that the Query graph has all of the pattern's elements.**	
0006	Satisfy the condition that all elements in the "pattern" list appear in the "query" list.	✔
0007	**Compare with the other pattern, which would not be presented in the Query and demonstrates the inequality**	
0008	By performing the WHERE ALL condition, both graphs should be unequal	✔
0009	**Check whether or not the duplicate elements (Sender POS, Receiver POS, and relationship type) of the "pattern" are present in the "query"**	
0010	Utilize an APOC procedure to collect duplicate elements from both the "pattern" and "query" lists.	✔
0011	Satisfy the condition that "query" contains all duplicate elements of "pattern"	✔
0012	Examine both "pattern" and "query" for duplicate elements that are not equivalent	✔
0013	Verify that none of the lists has duplicate elements; this should have no effect on the right results.	✔
0014	Confirm that no relationship conflicts occur when the same type of POS is replicated in the graph.	✖

Figure 5.20: Test cases

It works perfectly in the majority of cases, except for one, which involves a *relationship conflict*. As a result, I'll be modifying my Cypher query to address the issue.

5.3.3 Overcoming the problems from the previous cypher query

It is essential to resolve the relationship conflict that happens when the graph contains similar types of *POS*. But first, let's explore the basis of this relationship conflict.

What is relationship conflict?

Indeed, the built Cypher query will verify the existence of each outgoing relationship in a *saved pattern* graph by comparing it to the *Query* graph. However, if the *Query* graph or *saved pattern* graph contains the same kind of POS several times (for example, *NN* type POS is supplied twice), the comparison may result in a relationship conflict. To illustrate, I've included an example *Query* graph (inspired by Preposition Phrase pattern from [3]) and a sample pattern graph.

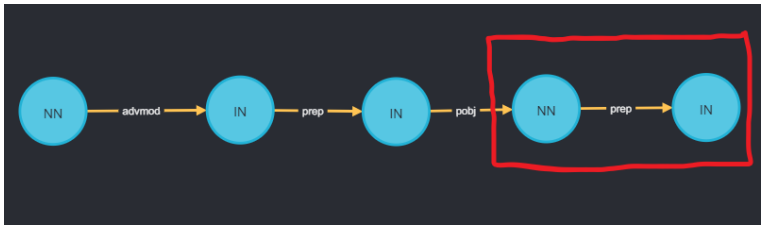

Figure 5.21: Sample Query graph

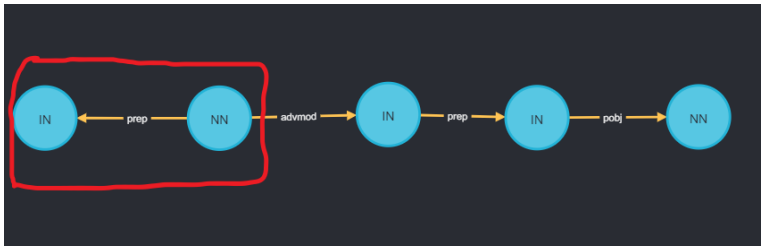

Figure 5.22: Pattern 6

When we look at the sample *Query* graph, we find that it contains two *NN* POS and three *IN* POS, as well as the relationships between them. Similarly, *pattern 6*, which is a sample pattern graph, contains two *NN* and three *IN* POS and relationships. As a consequence, when I compare this graph using the generated Cypher query, it meets the constraints and returns the result,

indicating that *pattern 6* matches the example *Query* graph. However, it should not be matched, because both graphs include *prep* relationship from *NN* to *IN* (which I have highlighted in both graphs), and while this is the common relationship between the two graphs, it is not in the precise structure. Because the sending node NN in the *Query* graph receives an incoming relationship of type *pobj* from another node, whereas the sending node NN in *pattern 6* does not get an incoming relationship. Thus, this demonstrates that both are not the same kind of one, which results in the conflict. However, the designed Cypher query is unaware of this.

To avoid this, I experimented with the concept of checking the sending node's incoming relation type only when the same kind of *POS* occurs many times in the same graph. Due to the fact that this results in relationship conflict. Thus, the above Cypher query collects the sender node's POS, the recipient node's POS, and the type of relationship between them. However, the amended Cypher query will return the incoming relationship type of the sender node as well. This indicates the Cypher query to check whether or not the Query graph contains the precise structure of the pattern graph.

As a result, I created a new Cypher query with certain improvements to prevent the conflicting relationships. I will describe each stage of the Cypher query in detail below.

- To begin, similar to the previous designed Cypher Query, I'm checking the outgoing relationship of each node in the pattern graph (in this case, pattern 6, a sample graph created for experimentation) by using the MATCH keyword.

  ```
  MATCH (a:pattern6)-[r1]->(b)
  ```

  Additionally, I'm using the OPTIONAL MATCH clause in the cypher statements as specified below. Indeed, the advantage of the OPTIONAL MATCH keyword is identical to that of the MATCH keyword, with the exception that if no MATCH is discovered for the specified criteria to be checked, it bypasses it by treating it as null. [79]

  ```
  OPTIONAL MATCH (a:pattern6)-[rr]->(b)
  OPTIONAL MATCH (i)-[ir]->(a)
  ```

  This is to examine separately the incoming relationship of a node that has an outgoing relationship with another node. For instance, I'm first verifying the outgoing relationship of a node in *pattern 6*, where *a* is the sender node, *b* is the receiver, and *rr* is the relationship type. Then, I verify the incoming relationship type of the sender node *a* to determine which node is its *head* node and what type of relationship they have, where *i* is the *head* node and *ir* is the type of relationship.

- Following that, as an update to the preceding Cypher query's aggregation process, utilizing *collect ()* in conjunction with the WITH keyword (as indicated in item 2). Additionally, I'm aggregating the incoming relationship type and storing it in a list named *incoming_pattern*.

```
WITH collect({SenderPOS:a.POS,relation:type(r1),
ReceiverPOS:b.POS}) AS pattern,
collect({Incoming:type(ir),SendPOS:a.POS,
relation:type(rr)}) AS incoming_pattern
```

- Then, as an update to the preceding Cypher query's collection mechanism within the CALL subquery for Query graph (as specified in item 3), I'm adding a Cypher statement to collect the incoming relationship type of a sender node and store it in a list named *incoming_query*. Additionally, because the sample Query graph is based on the *preposition phrase pattern*, I stored it in the database under the same name.

```
    CALL
{
    MATCH (n:PrepositionPhrase)-[r]->(p)
    OPTIONAL MATCH (n:PrepositionPhrase)-[rr]->(p)
    OPTIONAL MATCH (i)-[ir]->(n)
    RETURN collect ({SenderPOS:n.POS,relation:type(r),
    ReceiverPOS:p.POS}) AS query,
    collect({Incoming:type(ir),SendPOS:n.POS,
    relation:type(rr)}) AS incoming_query


}
```

- Then, in a separate CALL subquery, I'm collecting the sent node's POS property from both the *pattern* and the *Query* graph and saving them as POS_pattern and POS_query. Because these properties will be needed for more analysis to find out if there is relationship conflict in a graph.

```
CALL {
       MATCH (pos:" + label + ")
       RETURN collect(pos.POS) AS POS_pattern
     }
CALL {
       MATCH (pos:Query)
       RETURN collect(pos.POS) AS POS_query
     }
```

- Following that, I'm using the WITH keyword to manipulate all the gathered lists. To begin, I assigned a new variable to *ip* for the list of *incoming_pattern* and *iq* for the list of *incoming_query*. Because those lists must be reintroduced using the WITH keyword. Otherwise, I will be unable to access it in the future. Then, identical to the previous Cypher query, I looked for duplicate items in the list *pattern* and put them in a *dup_pattern* list, as well as duplicate elements in the list *query* and placed them in a separate list called *dup_query*. Additionally, I'm utilizing an APOC technique known as *apoc.coll.duplicates ()*. [80] This technique is identical to the other APOC procedure for detecting duplicate elements, except that it does not count the number of duplicate elements; it just finds them. Thus, I'm verifying whether the same kind of POS is present in the *pattern* or the *query* from the list that previously gathered the sender node's POS type from both graphs, and then saving the duplicate POS type presented in the pattern graph as a *dup_POS_pattern* and in the *query* graph as a *dup_POS_query*.

```
WITH incoming_pattern AS ip,
incoming_query AS iq,
apoc.coll.duplicates(POS_pattern) AS
dup_POS_pattern,
apoc.coll.duplicates(POS_query) AS
dup_POS_query,
apoc.coll.duplicatesWithCount(pattern) AS
dup_pattern,
apoc.coll.duplicatesWithCount(query) AS
dup_query
```

- Next, as similar to the previous cypher query (item 5 and item 6), I'm using Neo4j's predicate function WHERE ALL to determine whether or not the *pattern's* whole set of components (including duplicates) is provided in the *query*.

```
WHERE ALL(a IN pattern WHERE a IN query) AND
ALL(a IN dup_pattern WHERE a IN dup_query)
```

- Additionally, I'm determining whether the duplicate POS of nodes supplied in the collections *pattern* and the *query* are equal and recording the result in a collection named *Equal_POS*. This is accomplished through the use of the APOC procedure *apoc.coll.isEqualCollection()*, which compares the two collections and returns TRUE if they are identical. [81] For instance, in our scenario, the pattern graph has two POS of type *NN* and three POS of type *IN* (as represented in Figure 5.22), and the query graph contains the same types of POS for the same number of

times (illustrated in Figure 5.21). As a consequence, both of these graphs meet the equality criteria when using *apoc.coll.isEqualCollection().*

Then, using the *apoc.coll.containsAll()* procedure, I determine whether or not the *iq* collection includes all of the elements in *ip* and save the results to a collection named *Equal_Incoming*. Basically, I'm verifying to see if the *query* graph has the elements, including the sender node's incoming relationship type, that were presented in the *pattern* graph. For instance, if we consider our *pattern* graph (shown in Figure 5.22) in which an *IN* POS type node transmits a *prep* relationship type to another *IN* and receives an *advmod* type relationship from another node as an incoming relationship. Thus, if *Query* also has the same type, i.e., an *IN* node gets an *advmod* type relationship and sends a *prep* to another *IN* node, the relationship is valid and both nodes are in the identical structure. Similarly, it verifies all of the graph's sending nodes.

As a final condition, I'm using a WHERE condition to determine whether both graphs (pattern and query) contain the same type and number of POS; if they do, this will result in *relationship conflict*; to avoid this, the Cypher query must also check for the sender node's *incoming relationship type*; as a result, both graphs must have identical structure. That is why, in my constructed Cypher query, I specified using an AND operator that while *Equal_POS* is *true*, *Equal_Incoming* must also be *true.* On the other hand, if no graph contains the same kind and number of POS, there will be no relationship conflict, and therefore it doesn't matter if the verification of the sender node's incoming relationship type can be satisfied or not. As a result, I specified in my built Cypher Query utilizing an OR operator that if *Equal_POS* is false, and *Equal_Incoming* might be either *true* or *false.*

```
WITH apoc.coll.isEqualCollection(dup_POS_pattern,
dup_POS_query) AS Equal_POS,
apoc.coll.containsAll(iq,ip) AS Equal_Incoming
WHERE (Equal_POS = true AND Equal_Incoming = true)
OR (Equal_POS = false AND (Equal_Incoming = false
OR Equal_Incoming = true))
```

However, what if I set the condition that checks for the sender node's incoming relationship true at all times, despite the fact that there will be no POS of the same kind or number and no likelihood of relationship conflict?

Indeed, it may produce incorrect findings. Consider the following scenario: if we have a *pattern* similar to that illustrated in Figure 5.23, with just one sender node and no duplicate POS given, there will be no relationship conflict. On the other hand, if we compare this *pattern* to the

example *Query* graph in Figure 5.24, we see that it is really present there (I have highlighted it for illustration purposes), and the result should be matched. However, if I add a condition requiring that the sender node's incoming relation be fulfilled, the *pattern* graph will fail the condition and will not match the *Query*. However, this is the incorrect outcome. As a result, I included an incoming relationship type check of the sender node only when the nodes have the potential to cause a relationship conflict.

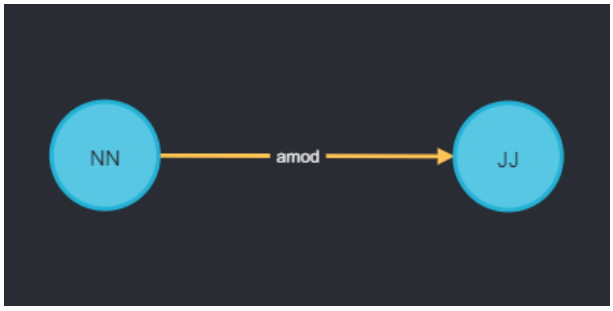

Figure 5.23: Example pattern

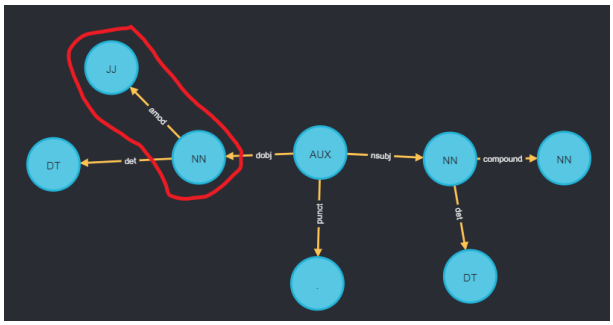

Figure 5.24: Example Query

- Finally, if all of the specified requirements are met and the pattern is legitimate, it becomes a subgraph of Query and is matched. As a result of the RETURN keyword, the value 1 will be printed.

```
RETURN 1
```

As a result, when I run this amended Cypher query directly from Neo4j's GUI, I received no return value since *pattern6* did not match the *Query*, but it is now a perfect result(illustrated in Figure 5.25). Indeed, I had previously obtained an incorrect return (as matched) due to a relationship conflict, but the modified Cypher query resolved the issue. Additionally, it works effectively and produces accurate answers for other patterns as well.

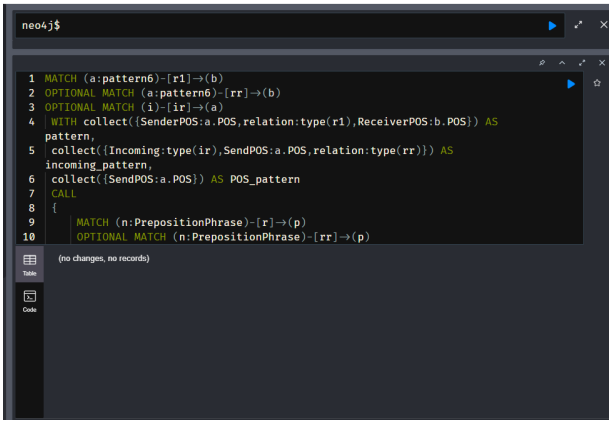

Figure 5.25: Screenshot of executing a modified Cypher query

Eventually, the updated Cypher query satisfies all test cases, including avoidance of relationship conflicts.

0001	**Check the outgoing relationship and collect the POS and relationship type of the sender and receiver nodes**	
0002	Examine the stored pattern's outgoing relationships and collect the POS (Sender & Receiver) and relationship type, then store them in a list named "pattern"	✔
0003	Examine the Query's outgoing relationships and collect the POS (Sender & Receiver) and relationship type, then store them in a list named "query"	✔
0004	Display both lists and check that they are appropriately stored	✔
0005	**Ascertain that the Query graph has all of the pattern's elements.**	
0006	Satisfy the condition that all elements in the "pattern" list appear in the "query" list.	✔
0007	**Compare with the other pattern, which would not be presented in the Query and demonstrates the inequality**	
0008	By performing the WHERE ALL condition, both graphs should be unequal	✔
0009	**Check whether or not the duplicate elements (Sender POS, Receiver POS, and relationship type) of the "pattern" are present in the "query"**	
0010	Utilize an APOC procedure to collect duplicate elements from both the "pattern" and "query" lists.	✔
0011	Satisfy the condition that "query" contains all duplicate elements of "pattern"	✔
0012	Examine both "pattern" and "query" for duplicate elements that are not equivalent	✔
0013	Verify that none of the lists has duplicate elements; this should have no effect on the right results.	✔
0014	Confirm that no relationship conflicts occur when the same type of POS is replicated in the graph.	✔

Figure 5.26: Satisfied test cases

Then, in accordance with this thesis need, it is time to build this Cypher query to execute dynamically in order to compare all the patterns in the database with the Query graph and return the matched patterns as results.

5.3.4 Dynamic way of checking

So far, I've illustrated how to statically match a *saved pattern* with a *Query* graph, i.e., by defining a specific *pattern* to be matched with *Query* manually. However, in this part, I will demonstrate how to dynamically execute a defined Cypher query based on the resources and information collected in the analysis and design part of dynamic execution (section 3.3.2.1).

First, as designed, I'm retrieving all of the *label* names associated with the graphs saved in the database through Neo4j's built-in procedure, which is invoked via a CALL clause and saves the label names via a YIELD clause.

Following that, I'm invoking the apoc.cypher.run() method with my statically defined Cypher query. Additionally, rather of manually checking the label names of the pattern graph, such as pattern 1, pattern 2, etc., I'm supplying the gathered label names from the YIELD clause. Thus, whenever the pattern label name should appear within the Cypher query, those locations are replaced with dynamic node labels.

For example:

```
MATCH (a:" + label + ")-[r1]->(b)
```

Therefore, this APOC procedure along with the Cypher query will access the *pattern* graphs stored in the database through the *label* name, check for a match with the *Query* graph, and ultimately deliver the *label* names of the graphs that are all matched with the *Query* graph.

However, how will it deliver the label name for the matched graph as a result?

Indeed, the Cypher query included within the **apoc.cypher run()** procedure will RETURN 1 if the *pattern* currently being verified matches the *Query* graph. If it is not found, no value is printed. Additionally, following this dynamic execution in the APOC procedure, I'm collecting the returned values using a YIELD clause, so that if the performed query produced any value, i.e., 1, if matched, that value will be collected and the *label* name of the *pattern* graph will be returned. If no value is given, the *pattern* has not been matched, and hence the *label* name is not returned.

```
CALL db.labels()
YIELD label
CALL apoc.cypher.run("MATCH (a:" + label + ")-[r1]->(b)
OPTIONAL MATCH (a:" + label + ")-[rr]->(b)
// Cypher is executing dynamically
.
.
.
//Checking pattern graph with Query for matching
// Print result when it is matched
RETURN 1",{})
YIELD value
RETURN label
```

Then, by performing the dynamic Cypher query in Neo4j's GUI, I obtained the label name for the matched pattern. I've included a screenshot in Figure 5.27.

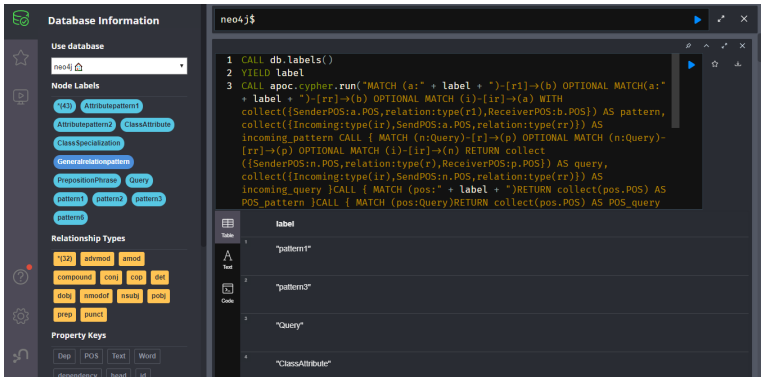

Figure 5.27: Screenshot of executing a dynamic Cypher query

Ultimately, the dynamic Cypher query is ready to get the Query(Sentence) graph's matching patterns. However, this must be integrated into the Python code in order to verify the sentence patterns received from TEMOS and return the matching patterns as results.

5.3.5 Incorporating Cypher query inside the Python function

The Cypher query can be easily integrated into Python code for execution through Py2neo's **run ()** method. I have previously used this function to insert queries within Python (section 5.2.3). As such, I'm going to add the designed dynamic cypher query into the *run ()* operation here as well to obtain matching patterns as results.

Actually, I need to incorporate this operation into the Python code's **get_matched_patterns** function (section 5.3). Additionally, I'm inserting this Cypher execution operation immediately following the iterative process of obtaining the sentence pattern from TEMOS and loading it into the database for analysis. Thus, after the sentence pattern (Query) is stored in the database, this Cypher matching query will connect with the Neo4j server and perform the necessary matching operations between the *Query* and other specified patterns in the database. However, the question of delivering the results of the pattern matching to Python may arise.

Indeed, such behavior is feasible. Because I've assigned this **run ()** operation to a variable named *matched* just for this reason. Thus, after this query is executed, the **run ()** operation will create results in the form of a stream of *records*, which will be stored in the variable named *matched*. Actually, records are an ordered collection of values that are categorized according to key-value

pairs. [51] In this situation, the value is the *name* of a label, and the key is the *label*.

After capturing the records in a *matched* variable, I iterate over them, adding each record to an empty list depending on its key, *label*. Thus, all returned label names for matched patterns are now placed in the empty list called *match = []*.

Finally, by returning the list *match*, the matched pattern graphs' label names will display in the Python console.

I've included the code for the described procedure below.

```
# An empty list to capture returned results
match = []
matched = self.graph.run( #Cypher query will execute)

#iterating through the returned records
for record in matched:
    # appending each record based on it's key "label"
    #inside the empty list
    match.append(record["label"])

# returning the label name as results
return match
```

5.3.5.1 Deletion of Query graph after pattern matching

Once the pattern matching is complete and the results are obtained, the Query graph should be erased from the database. Additionally, it should create room for the arrival of a new Query(sentence) for pattern matching, and the process will repeat. As a result, I developed the following Cypher query to conduct this delete action.

```
MATCH (n:Query)
OPTIONAL MATCH(n)-[r]-()
DELETE n,r
```

This delete query will look for the graph labeled *Query*. Then, because the OPTIONAL MATCH keyword is used, it will verify that all nodes in the *Query* graph are related to one another. If yes, it will first destroy the relationships, and then the nodes. If the nodes are not linked, it just deletes them all. Finally, the database will be emptied of the whole Query graph.

Moreover, as like other Cypher queries, I incorporated this delete query as well inside the graph.run() operation in Python code as given below.

```
self.graph.run("MATCH(n:Query) "
               "OPTIONAL MATCH(n)-[r]-() "
               "DELETE n,r")
```

5.4 Summary

In conclusion, let's review my total implementation process. To begin, in section 5.2, I demonstrated how to store linguistic patterns in Neo4j both directly via the Neo4j graphical user interface (section 5.2.1) and primarily using Python code (section 5.2.2). I detailed the Python function that was developed for the storage purpose, as well as its behavior.

After that, in section 5.3, I detailed and showed how to retrieve sentence patterns and load them as an input to the Neo4j database as a graph.

Next, I developed a Cypher query in section 5.3.1 to do pattern matching between the *pattern* and the *Query(sentence)* graph. Then, in section 5.3.2, I designed a test case to see whether or not the built Cypher query is functional. As a consequence, I resolved the issues in section 5.3.3 by modifying the Cypher query to function properly.

In section 5.3.4, I enhanced my statically written Cypher query by dynamically matching patterns.

Finally, in section 5.3.5, I integrated my completely finished Cypher query into the Python code and executed it to return the matching patterns. Additionally, in section 5.3.5.1, I explained the construction of the delete query that would destroy the *Query* graph from the database after pattern matching is complete.

Chapter 6

Testing

It is now necessary to test the fully implemented code to check that it functions properly and complies with the thesis criteria. As a result, I'll discuss the testing of the code and provide test results in this chapter.

6.1 Testing in Pytest

I'm using *Pytest*, a testing tool that enables me to rapidly and successfully test the generated Python code's operations for accurate results. Besides, I have deployed this testing tool as mentioned in the configuration stage (section 4.2.3).

To begin, what exactly should be evaluated?

1. The developed Python code should properly accept the sentence pattern sent as a list and generate the graph in Neo4j by iterating over the instance attributes included inside the list.
2. Then, pattern matching between the received Query (sentence) and other predetermined patterns stored in the database should be performed, and the matched patterns should be presented as results, and they should be the correct ones.
3. Finally, the database should be cleared of the examined Query (sentence). Additionally, it should be prepared to take another Query(sentence) for analysis, at which point the process will repeat. Thus, it should accept any Query(sentence) and analyze it, returning accurate results.

Following that, I generated a separate Python file containing the test functions to test the aforementioned actions. And each function returns the sentence to be analyzed as a list of objects, with each element pointing to an instance attribute of the *Token* class, which contains the initializer for initializing the attributes (text, pos, dep, and head) required for textual patterns,

as demonstrated in the section 5.2.2. Additionally, I created these sentence patterns in accordance with the papers [2] and [3]. Moreover, in these papers, some of the example sentences don't have grammatical inspections, so I used a Spacy-NLP tool [13] to get *pos tags* and *dependencies* for the texts. Based on that, I made a sentence pattern for testing.

After that, I created an object for the class *GraphPatternsChecker()* that I imported from the *graph_patterns_checker.python file*. Then, using the created object, I'm passing the sentence list as an argument to run the function named *get_matched_patterns.* Indeed, this function is responsible for receiving the sentence and loading it into Neo4j, as described in section 5.3. Moreover, the results will be generated once *get_matched_patterns* completes its process. So, to capture those results, I'm calling that function from within the *matched_patterns* variable. As a result, when *get_matched_patterns()* returns a result, it will be stored in this variable.

Now, I'm going to use Python's *assert* keyword to verify that the returned results are accurate. This is to ensure that the condition is satisfied and to return true if it is; else, an *AssertionError* is thrown. Thus, by using the assert keyword, I'm ensuring that the collected results of the *matched_* patterns reflect the desired results. If it passes the test, it indicates that the constructed code is accurate and provides the expected outcomes.

However, what if the *matched_patterns* and the desired results are identical but one of them has elements in a different order?

In fact, it would cause an *AssertionError*. To avoid this, equality checks should be performed regardless of the order of the elements. As a consequence, I utilized the built-in Python method *frozenset()*. Thus, if I used this to compare two frozenset collections, it would check for equality regardless of the elements order.

I've included a code sample below to illustrate my overall explanation.

```
def test_sentence():

#Sentence patterns as list of objects
sentence = [
.
.
# Attributes necessary for textual patterns are present.
.
]

graph_patterns_checker = GraphPatternsChecker()
matched_patterns = \
graph_patterns_checker.get_matched_patterns(sentence)
```

```
assert frozenset(matched_patterns) == \
       frozenset(['pattern1', 'pattern3', 'Query',
                  'ClassAttribute'])
```

Similarly, I developed ten distinct sorts of test functions that each hold a unique sentence pattern. Then, using the pytest command, I ran the tests.py file that contains all of these test functions.

As a result, all test functions passed, indicating that the Python code was written correctly and produces accurate results. I've included a screenshot of a run I performed in my Python terminal below.

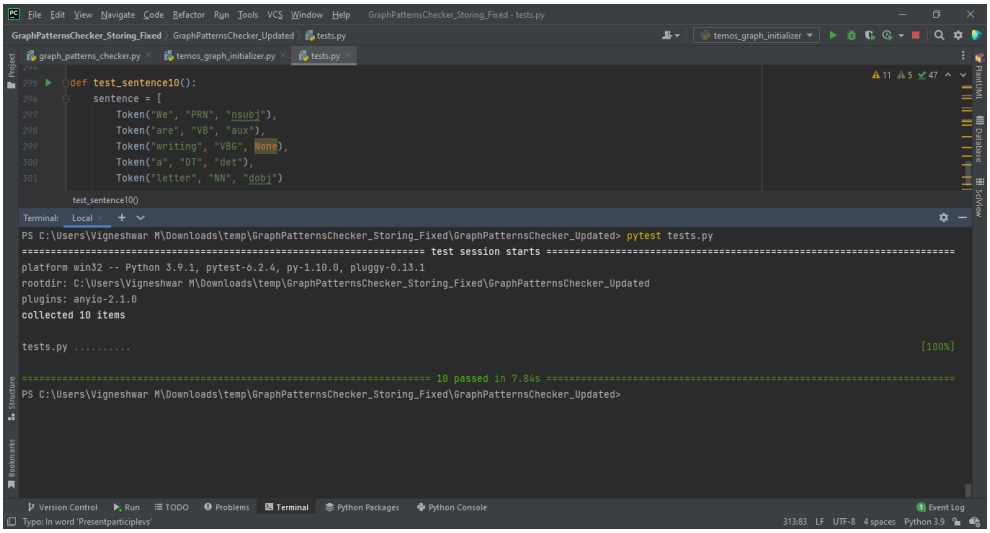

Figure 6.1: Screenshot of running a pytest

6.2 Summary

In summary, the test findings satisfy the following valid requirements specified at the beginning of this testing chapter:

- The Python code successfully accepted the sentence patterns, which were supplied as a list, and built the Neo4j graph by iterating over the list's attributes.
- Successful pattern matching between the received *Query(sentence)* and *predefined patterns*, with accurate results generated.
- Finally, the investigated *Query(sentence)* has been deleted, as evidenced by the succession of all test functions, indicating that the analyzed *Query* was deleted to make room for another sentence.

Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

This thesis focused on constructing a Neo4j database for graph-based storage of linguistic patterns. It incorporates both predefined patterns and sentence patterns(Query) obtained from a TEMOS through a shared Python interface. Then, using a Cypher query in Neo4j, pattern matching was performed between the received sentence pattern(Query) and predetermined patterns, and the results were successfully created. As a result, I conclude that these matched patterns are ready to be used in additional text mining operations, namely to aid in the process of semantic enrichment and the discovery of inaccuracies in a textual requirements specification. Additionally, the implemented Neo4j database and Python program can be used to receive any textual linguistic pattern with its lexical content, transform it to structural content as a graph in Neo4j, and perform pattern matching using a dynamically designed Cypher query that compares the received sentence to all predefined linguistic patterns and generates the pattern name for all matched subgraphs of the received sentence pattern.

7.1.1 Assignment completion

I have fulfilled all of the prerequisites for this master's thesis, which are shown below:

1. I introduced the Neo4j graph database and described why it is useful, as well as how it is possible to store graphs using the Cypher Query Language, on which Neo4j is based.

2. I gave a brief introduction to the linguistic patterns that were dealt with in this thesis. Additionally, I discussed the linguistic patterns'

basis, which is Natural Language Processing (NLP), and how these linguistic patterns are employed in requirement specifications to discover inaccuracies in natural language text.

3. As a core part of this thesis, I designed and implemented the Neo4j graph database for storing linguistic patterns generated by Python code (which also stores cypher queries), utilizing a Py2neo interface to interact Python and Neo4j. Additionally, I demonstrated another method for storing linguistic patterns in Neo4j by constructing a Cypher Query directly in Neo4j's graphical user interface.

4. Following that, I implemented a Python shared interface that enables TEMOS software to provide sentence patterns (Query) for analysis, which are then transferred to Neo4j as a graph through the Python to Neo4j interface. After that, the constructed dynamic cypher query was used to execute pattern matching, resulting in the generation of matched patterns.

5. Finally, all of these operations were verified using pytest to confirm that the Python code developed, including the Cypher query, performed as expected and delivered accurate results. As a result, it can accept any sentence that emanates from TEMOS, do pattern matching, and deliver accurate results.

7.2 Future Work

The database that was created lays the foundation for future enhancements. The present solution involves retrieving one Query(sentence) from TEMOS at a time, analyzing it, and then deleting it to create space for another Query(sentence). However, this procedure may be modified to receive numerous inputs concurrently and do pattern matching analysis and produce results. This may be accomplished by the use of a Python iteration with a dynamic range to take many Query (sentences) concurrently, or with developments in Cypher querying through the use of the efficient APOC procedures well-known for dynamic operations.

Bibliography

[1] Requirements Specification[online]. [Cited 2021-10-23]. Available from: https://visuresolutions.com/requirements-specification/

[2] Šenkỳr, D.; Kroha, P. Patterns in textual requirements specification. In *Proceedings of the 13th International Conference on Software Technologies, Porto, Portugal*, 2018, pp. 197–204.

[3] Šenkỳř, D.; Kroha, P. Patterns of ambiguity in textual requirements specification. In *Rocha, A. et al: Proceedings of WorldCIST'19 – World Conference on Information Systems and Technologies, Advances in Intelligent Systems and Computing Nr. 930, New Knowledge in Information Systems and Technologies*, volume 1, Springer, 2019, pp. 886–895.

[4] Šenkỳr, D.; Kroha, P. Problem of incompleteness in textual requirements specification. In *Proceedings of the 14th International Conference on Software Technologies, Porto, Portugal*, volume 1: ICSOFT, SciPress, 2019, ISBN 978-989-758-379-7, pp. 323–330.

[5] Šenkýř., D.; Kroha., P. Problem of Inconsistency in Textual Requirements Specification. In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,*, INSTICC, SciTePress, 2021, ISBN 978-989-758-508-1, ISSN 2184-4895, pp. 213–220, doi:10.5220/0010421602130220.

[6] Šenkýř, D.; Kroha, P. Problem of Semantic Enrichment of Sentences Used in Textual Requirements Specification. In *Advanced Information Systems Engineering Workshops*, edited by A. Polyvyanyy; S. Rinderle-Ma, Cham: Springer International Publishing, 2021, ISBN 978-3-030-79022-6, pp. 69–80.

[7] What is Natural Language Processing?[online]. [Cited on 2021-10-29]. Available from: https://www.ibm.com/cloud/learn/natural-language-processing

[8] NLP vs. NLU vs. NLG: the differences between three natural language processing concepts [online]. Nov. 2020, [cited on 2021-10-31]. Available from: https://www.ibm.com/blogs/watson/2020/11/nlp-vs-nlu-vs-nlg-the-differences-between-three-natural-language-processing-concepts/

[9] Marshall, C. What is named entity recognition (NER) and how can I use it?[online]. June 2020, [Cited on 2021-10-31]. Available from: https://medium.com/mysuperai/what-is-named-entity-recognition-ner-and-how-can-i-use-it-2b68cf6f545d

[10] Liddy, E. Natural language processing in Encyclopedia of Library and Information Science 2nd ed., New York: Marcel Decker. 2001.

[11] Español, C. K. What are the different levels of NLP?[online]. Apr. 2020, [Cited on 2021-11-01]. Available from: https://medium.com/@CKEspanol/what-are-the-different-levels-of-nlp-how-do-these-integrate-with-information-retrieval-c0de6b9ebf61

[12] Natural Language Processing Tools and Libraries in 2021[online]. [Cited on 2021-11-02]. Available from: https://theappsolutions.com/blog/development/nlp-tools/

[13] spaCy · Industrial-strength Natural Language Processing in Python[online]. [Cited on 2021-12-26]. Available from: https://spacy.io/

[14] Elgabry, O. Requirements Engineering — Requirements Specification (Part 3) [online]. Sept. 2017, [Cited on 2021-11-04]. Available from: https://medium.com/omarelgabrys-blog/requirements-engineering-elicitation-analysis-part-5-2dd9cffafae8

[15] Bhonde, S.; Paikrao, R.; et al. Text association analysis and ambiguity in text mining. In *AIP Conference Proceedings*, volume 1324, American Institute of Physics, 2010, pp. 204–206.

[16] Vimalraj, T.; Seema, B. Identification of Ambiguity in Requirement Specification using Multilingual Word Sense. In *International Journal of Advanced Research in Computer and Communication Engineering*, *Issue 6*, volume Vol. 5, June 2016, ISSN 2278-1021 [online].

[17] Gleich, B.; Creighton, O.; et al. Ambiguity Detection: Towards a Tool Explaining Ambiguity Sources. In *Requirements Engineering: Foundation for Software Quality*, edited by R. Wieringa; A. Persson, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN 978-3-642-14192-8, pp. 218–232.

[18] Spanoudakis, G.; Zisman, A. Inconsistency Management in Software Engineering: Survey and Open Research Issues. *Handbook of Software Engineering and Knowledge Engineering*, 11 2000, doi:10.1142/9789812389718_0015.

[19] Silva, A. Linguistic Patterns and Linguistic Styles for Requirements Specification (I): An Application Case with the Rigorous RSL/Business-Level Language. 07 2017, pp. 1–27, doi:10.1145/3147704.3147728.

[20] Full List Of Annotators[online]. [Cited on 2021-11-17]. Available from: https://stanfordnlp.github.io/CoreNLP/annotators.html

[21] The Stanford Natural Language Processing Group[online]. [Cited on 2021-11-17]. Available from: https://nlp.stanford.edu/software/tagger.html

[22] What is a Graph Database? - Developer Guides [online]. [Cited on 2021-11-09]. Available from: https://neo4j.com/developer/graph-database/

[23] Graph Database Use Cases. Available from: https://neo4j.com/use-cases/

[24] Di Maro, M.; Valentino, M.; et al. Graph databases for designing high-performance speech recognition grammars. In *IWCS 2017—12th International Conference on Computational Semantics—Short papers*, 2017.

[25] Cypher Query Language - Developer Guides. Available from: https://neo4j.com/developer/cypher/

[26] openCypher · openCypher [online]. [Cited on 2021-11-16]. Available from: http://opencypher.org/

[27] Getting Started with Cypher - Developer Guides [online]. [Cited on 2021-11-16]. Available from: https://neo4j.com/developer/cypher/intro-cypher/

[28] Querying with Cypher - Developer Guides[online]. [Cited on 2021-11-20]. Available from: https://neo4j.com/developer/cypher/querying/

[29] Filtering Query Results - Developer Guides[online]. [Cited on 2021-11-20]. Available from: https://neo4j.com/developer/cypher/filtering-query-results/

[30] ORDER BY - Neo4j Cypher Manual[online]. [Cited on 2021-11-20]. Available from: https://neo4j.com/docs/cypher-manual/4.3/clauses/order-by/

[31] Updating with Cypher - Developer Guides [online]. [Cited on 2021-11-21]. Available from: https://neo4j.com/developer/cypher/updating/

[32] Comparing SQL with Cypher - Developer Guides [online]. [Cited on 2021-11-28]. Available from: https://neo4j.com/developer/cypher/guide-sql-to-cypher/

[33] User Defined Procedures and Functions - Developer Guides[online]. [Cited on 2021-12-05]. Available from: https://neo4j.com/developer/cypher/procedures-functions/

[34] Neo4j Labs - Neo4j Labs [online]. [Cited on 2021-12-05]. Available from: https://neo4j.com/labs/

[35] Procedures & Functions - APOC Documentation[online]. [Cited on 2021-12-06]. Available from: https://neo4j.com/labs/apoc/4.1/overview/

[36] Drivers & Language Guides - Developer Guides[online]. [Cited on 2021-11-21]. Available from: https://neo4j.com/developer/language-guides/

[37] Bolt Protocol[online]. [Cited on 2021-11-21]. Available from: https://boltprotocol.org/

[38] 4.2. Client applications - Chapter 4. Drivers[online]. [Cited on 2021-11-23]. Available from: https://neo4j.com/docs/developer-manual/current/drivers/client-applications/

[39] API Documentation — Neo4j Python Driver 4.3 [online]. [Cited on 2021-11-23]. Available from: https://neo4j.com/docs/api/python-driver/current/api.html#session

[40] API Documentation — Neo4j Python Driver 4.3 [online]. [Cited on 2021-11-23]. Available from: https://neo4j.com/docs/api/python-driver/current/api.html#transaction

[41] Introduction - HTTP API [online]. [Cited on 2021-11-22]. Available from: https://neo4j.com/docs/http-api/4.3/introduction/

[42] Transaction flow - HTTP API [online]. [Cited on 2021-11-22]. Available from: https://neo4j.com/docs/http-api/4.3/actions/transaction-flow/

[43] Postman API Platform [online]. [Cited on 2021-11-22]. Available from: https://www.postman.com/product/what-is-postman/

[44] Welcome to Paradise Paper Search App's Django + Neomodel Tutorial! — Paradise Paper Search 1 documentation [online]. [Cited on 2021-11-25]. Available from: https://neo4j-examples.github.io/paradise-papers-django/

[45] The Py2neo Handbook — py2neo 2021.1 [online]. [Cited on 2021-11-25]. Available from: https://py2neo.org/2021.1/#core-graph-api

[46] Connection profiles — py2neo 2021.1[online]. [Cited on 2021-11-25]. Available from: https://py2neo.org/2021.1/profiles.html#connection-profiles

[47] py2neo — py2neo 2021.1. [Cited on 2021-11-27]. Available from: https://py2neo.org/2021.1/_modules/py2neo.html#ConnectionProfile{[online]}

[48] Workflow — py2neo 2021.1. [Cited on 2021-11-28]. Available from: https://py2neo.org/2021.1/workflow.html#graphservice-objects{[online]}

[49] Workflow — py2neo 2021.1[online]. [Cited on 2021-11-29]. Available from: https://py2neo.org/2021.1/workflow.html#py2neo.Graph

[50] Workflow — py2neo 2021.1 [online]. [Cited on 2021-11-29]. Available from: https://py2neo.org/2021.1/workflow.html#transaction-objects

[51] py2neo.cypher – Cypher Execution — py2neo 2021.1 [online]. [Cited on 2021-11-29]. Available from: https://py2neo.org/2021.1/cypher/index.html#record-objects

[52] 2. py2neo.database – Graph Databases — The Py2neo v4 Handbook[online]. [Cited on 2021-11-29]. Available from: https://py2neo.org/v4/database.html#py2neo.database.Graph.run

[53] py2neo.data – Graph data types — py2neo 2021.1[online]. [Cited on 2021-11-30]. Available from: https://py2neo.org/2021.1/data/index.html#py2neo.data.Subgraph

[54] Node and relationship matching — py2neo 2021.1[online]. [Cited on 2021-11-30]. Available from: https://py2neo.org/2021.1/matching.html#node-matching

[55] Errors — py2neo 2021.1[online]. 2021-12-01. Available from: https://py2neo.org/2021.1/errors.html

[56] apoc.create.relationship - APOC Documentation[online]. [Cited 2021-12-06]. Available from: https://neo4j.com/labs/apoc/4.1/overview/apoc.create/apoc.create.relationship/

[57] apoc.merge.relationship - APOC Documentation [online]. [Cited on 2021-12-06]. Available from: https://neo4j.com/labs/apoc/4.1/overview/apoc.merge/apoc.merge.relationship/

[58] apoc.cypher.run - APOC Documentation. Available from: https://neo4j.com/labs/apoc/4.1/overview/apoc.cypher/apoc.cypher.run/

[59] CALL procedure - Neo4j Cypher Manual[online]. [Cited on 2021-12-09]. Available from: https://neo4j.com/docs/cypher-manual/4.4/clauses/call/

[60] Neo4j Supported Versions - Knowledge Base[online]. [Cited on 2021-12-10]. Available from: https://neo4j.com/developer/kb/neo4j-supported-versions/

[61] Installation - Operations Manual[online]. [Cited on 2021-12-10]. Available from: https://neo4j.com/docs/operations-manual/4.4/installation/

[62] Neo4j Desktop User Interface Guide - Developer Guides [online]. [Cited on 2021-12-10]. Available from: https://neo4j.com/developer/neo4j-desktop/

[63] Neo4j Download Center - Neo4j Graph Database Platform[online]. [Cited on 2021-12-10]. Available from: https://neo4j.com/download-center/#desktop

[64] Installation - APOC Documentation[online]. [Cited on 2021-12-10]. Available from: https://neo4j.com/labs/apoc/4.1/installation/

[65] Download Python[online]. [Cited on 2021-12-10]. Available from: https://www.python.org/downloads/

[66] PyCharm: the Python IDE for Professional Developers by JetBrains[online]. [Cited on 2021-12-11]. Available from: https://www.jetbrains.com/pycharm/

[67] Project Jupyter[online]. 2021-12-11. Available from: https://www.jupyter.org

[68] pip: The PyPA recommended tool for installing Python packages.[online]. [Cited on 2021-12-10]. Available from: https://pip.pypa.io/

[69] py2neo: Python client library and toolkit for Neo4j[online]. [Cited on 2021-12-10]. Available from: https://py2neo.org/

[70] The Py2neo Handbook — py2neo 2021.1[online]. [Cited on 2021-12-10]. Available from: https://py2neo.org/2021.1/#releases-versioning

[71] Installation and Getting Started — pytest documentation[online]. [Cited on 2021-12-11]. Available from: https://docs.pytest.org/en/6.2.x/getting-started.html

[72] pytest: pytest: simple powerful testing with Python[online]. [Cited on 2021-12-11]. Available from: https://docs.pytest.org/en/latest/

[73] apoc.merge.relationship - APOC Documentation[online]. [Cited on 2021-12-17]. Available from: https://neo4j.com/labs/apoc/4.1/overview/apoc.merge/apoc.merge.relationship/

[74] Aggregating functions - Neo4j Cypher Manual[online]. [Cited on 2021-12-18]. Available from: https://neo4j.com/docs/cypher-manual/4.4/functions/aggregating/

[75] WITH - Neo4j Cypher Manual[online]. [Cited on 2021-12-18]. Available from: https://neo4j.com/docs/cypher-manual/4.4/clauses/with/

[76] Predicate functions - Neo4j Cypher Manual[online]. [Cited on 2021-12-18]. Available from: https://neo4j.com/docs/cypher-manual/4.4/functions/predicate/

[77] apoc.coll.duplicatesWithCount - APOC Documentation[online]. [Cited on 2021-12-18]. Available from: https://neo4j.com/labs/apoc/4.1/overview/apoc.coll/apoc.coll.duplicatesWithCount/

[78] A Test Plan Tool for simpler Test Case Management | Testpad[online]. [Cited on 2021-12-20]. Available from: https://ontestpad.com/

[79] OPTIONAL MATCH - Neo4j Cypher Manual[online]. [Cited on 2021-12-20]. Available from: https://neo4j.com/docs/cypher-manual/4.4/clauses/optional-match/

[80] apoc.coll.duplicates - APOC Documentation[online]. [Cited on 2021-12-20]. Available from: https://neo4j.com/labs/apoc/4.2/overview/apoc.coll/apoc.coll.duplicates/

[81] apoc.coll.isEqualCollection - APOC Documentation[online]. [Cited on 2021-12-20]. Available from: https://neo4j.com/labs/apoc/4.2/overview/apoc.coll/apoc.coll.isEqualCollection/

Appendix A

List of Acronyms

APOC Awesome Procedures on Cypher
API Application Programming Interface
ACID Atomicity Consistency Isolation and Durability
CQL Cypher Query Language
CRUD Create Read Update Delete
GUI Graphical User Interface
HTTP Hyper Text Transfer Protocol
IDE Integrated Development Environment
JSON JavaScript Object Notation
NLP Natural Language Processing
NLTK Natural Language Toolkit
POS Parts of Speech Tagging
PyPI Python Package Index
RDF Resource Description Framework
SQL Structured Query Language
SSL Socket Secure Layer
TCP Transmission Control Protocol
TLS Transport Layer Security
TEMOS Textual Modelling System
UML Unified Modeling Language
URI Uniform Resource Identifier

Appendix B

Code

B.1 graph_patterns_checker.py

```
from py2neo import Graph, Node, Relationship


class Token:
    """
    This is a class for accessing through the sentence list.

    Attributes:
        text(str): The words in a sentence.
        pos(str): The part of speech tagging of a word.
        dep(str): The dependencies between words.

    """

    def __init__(self, text: str, pos: str, dep: str) -> None:
        """
        The constructor for Token class.
        :param text: The words in a sentence.
        :param pos: The part of speech tagging of a word.
        :param dep: The dependencies between words.
        """
        self.text = text
        self.pos = pos
        self.dep = dep
        self.head = None


class GraphPatternsChecker:
    """
    This is class to check a graph patterns for it's matching.
    """

    def __init__(self) -> None:
        """
```

```
35          The constructor to initiate the graph database connection
36          """
37          self.graph = Graph("bolt://localhost:7687", auth=("neo4j"
    , "123"))
38
39      def create_relationship(self, name: str):
40
41          self.graph.run("MATCH(p:" + name + "),(s:" + name + ")"
42                         "WHERE p.id = s.head WITH p, s "
43                         "CALL apoc.merge.relationship(p, "
44                         "(toString(s.Dep)), {}, {}, s, {}) "
45                         "YIELD rel RETURN p, s")
46
47      def insert_patterns(self, name: str, pattern: list[Token]) ->
     None:
48          """
49          A function to take defined patterns and store it
    dynamically based on relationships.
50
51          Parameters"
52              name: under which label name the patterns should be
    stored.
53              pattern(list): list of pattern's properties and
    relationships
54          """
55
56          for i in range(len(pattern)):
57
58              if pattern[i].head is not None:
59                  sender_node = pattern.index(pattern[i].head)
60
61              else:
62                  sender_node = None
63
64              node = Node(name, id=i, POS=pattern[i].pos, Word=
    pattern[i].text,
65                          Dep=pattern[i].dep,
66                          head=sender_node)
67
68              self.graph.merge(node, name, "id")
69              self.create_relationship(name)
70
71      def get_matched_patterns(self, sentence: list[Token]) -> list
    [str]:
72          """
73          A function to takes in a sentence and returns the matched
     patterns
74
75          Parameters:
76              sentence(list): The list of tokens.
77
78          Returns:
79              list(str): The list of matched patterns.
80          """
```

```
81
82          for k in range(len(sentence)):
83
84              if sentence[k].head is not None:
85                  head_value = sentence.index(sentence[k].head)
86
87              else:
88                  head_value = None
89
90              node = Node("Query", id=k, POS=sentence[k].pos, Word=
    sentence[k].text,
91                          Dep=sentence[k].dep,
92                          head=head_value)
93              self.graph.merge(node, "Query", "id")
94              self.create_relationship("Query")
95
96          match = []
97          matched = self.graph.run("""CALL db.labels()
98                          YIELD label
99                          CALL apoc.cypher.run("MATCH (a:" + label +
     ")-[r1]->(b)
100                         OPTIONAL MATCH (a:" + label + ")-[rr]->(b)
101                         OPTIONAL MATCH (i)-[ir]->(a)
102                         WITH collect({SenderPOS:a.POS,relation:
    type(r1),ReceiverPOS:b.POS}) AS pattern,
103                         collect({Incoming:type(ir),SendPOS:a.POS,
    relation:type(rr)}) AS incoming_pattern
104                         CALL { MATCH (n:Query)-[r]->(p)
105                         OPTIONAL MATCH (n:Query)-[rr]->(p)
    OPTIONAL MATCH (i)-[ir]->(n)
106                         RETURN collect ({SenderPOS:n.POS,relation:
    type(r),ReceiverPOS:p.POS}) AS query,
107                         collect({Incoming:type(ir),SendPOS:n.POS,
    relation:type(rr)}) AS incoming_query }
108                         CALL { MATCH (pos:" + label + ") RETURN
    collect(pos.POS) AS POS_pattern }
109                         CALL { MATCH (pos:Query) RETURN collect(
    pos.POS) AS POS_query }
110                         WITH incoming_pattern AS ip,incoming_query
     AS iq,
111                         apoc.coll.duplicates(POS_pattern) AS
    dup_POS_pattern,
112                         apoc.coll.duplicates(POS_query) AS
    dup_POS_query,
113                         apoc.coll.duplicatesWithCount(pattern) AS
    dup_pattern,
114                         apoc.coll.duplicatesWithCount(query) AS
    dup_query
115                         WHERE ALL(a IN pattern WHERE a IN query)
    AND ALL(a IN dup_pattern WHERE a IN dup_query)
116                         WITH apoc.coll.isEqualCollection(
    dup_POS_pattern,dup_POS_query) AS Equal_POS,
117                         apoc.coll.containsAll(iq,ip) AS
    Equal_Incoming
```

```
118                         WHERE (Equal_POS = true AND Equal_Incoming
    = true) OR
119                         (Equal_POS = false AND (Equal_Incoming =
   false OR Equal_Incoming = true))
120                         RETURN 1",{})
121                         YIELD value
122                         RETURN label""")
123
124         for record in matched:
125             match.append(record["label"])
126             self.graph.run("MATCH(n:Query)
127             OPTIONAL MATCH(n)-[r]-() DELETE n,r")
128
129         return match
```

B.2 temos_graph_initializer

```
1  from graph_patterns_checker import GraphPatternsChecker, Token
2
3
4  class TemosGraphInitializer:
5      """This is a class to store a predefined linguistics patterns
       in a database."""
6
7      def initialize(self) -> None:
8          pattern_1 = [
9              Token("NN", "NN", "nsubj"),
10             Token("AUX", "AUX", None),
11             Token("NN", "NN", "dobj")
12         ]
13
14         pattern_1[0].head = pattern_1[1]
15         pattern_1[2].head = pattern_1[1]
16
17         pattern_2 = [
18             Token("NN", "NN", "nsubj"),
19             Token("VB", "VB", None),
20             Token("NN", "NN", "dobj")
21         ]
22
23         pattern_2[0].head = pattern_2[1]
24         pattern_2[2].head = pattern_2[1]
25
26         pattern_3 = [
27             Token("JJ", "JJ", "amod"),
28             Token("NN", "NN", None)
29
30         ]
31
32         pattern_3[0].head = pattern_3[1]
33
34         pattern_4 = [
```

```
            Token("JJ", "JJ", "amod"),
            Token("NN", "NN", None),
            Token("DT", "DT", "det")

        ]
        pattern_4[0].head = pattern_4[1]
        pattern_4[2].head = pattern_4[1]

        pattern_5 = [
            Token("DT", "DT", "det"),
            Token("NN", "NN", "dobj"),
            Token("VB", "VB", None),
            Token("NN", "NN", "nsubj"),
            Token("DT", "DT", "det")

        ]
        pattern_5[0].head = pattern_5[1]
        pattern_5[1].head = pattern_5[2]
        pattern_5[3].head = pattern_5[2]
        pattern_5[4].head = pattern_5[3]

        class_attr = [
            Token("NN", "NN", "compound"),
            Token("NN", "NN", None)
        ]

        class_attr[0].head = class_attr[1]

        class_special = [
            Token("NN", "NN", "nsubj"),
            Token("VB", "VB", "cop"),
            Token("NN", "NN", "compound"),
            Token("NN", "NN", None),
            Token("NN", "NN", "conj"),
            Token("NN", "NN", "compound")
        ]

        class_special[0].head = class_special[3]
        class_special[1].head = class_special[3]
        class_special[2].head = class_special[3]
        class_special[4].head = class_special[3]
        class_special[5].head = class_special[4]

        Attr_1 = [
            Token("NN", "NN", "nsubj"),
            Token("VB", "VB", None),
            Token("NN", "NN", "dobj"),

        ]
        Attr_1[0].head = Attr_1[1]
        Attr_1[2].head = Attr_1[1]

        Attr_2 = [
            Token("NN", "NN", "nmodof"),
```

```
89              Token("NN", "NN", None)
90          ]
91
92          Attr_2[0].head = Attr_2[1]
93
94          General_relation = [
95              Token("NN", "NN", "nsubj"),
96              Token("VB", "VB", None),
97              Token("NN", "NN", "compound"),
98              Token("NN", "NN", "dobj")
99          ]
100         General_relation[0].head = General_relation[2]
101         General_relation[2].head = General_relation[3]
102         General_relation[3].head = General_relation[1]
103
104         Preposition_Phrase = [
105             Token("NN", "NN", None),
106             Token("IN", "IN", "advmod"),
107             Token("IN", "IN", "prep"),
108             Token("NN", "NN", "pobj"),
109             Token("IN", "IN", "prep")
110
111         ]
112         Preposition_Phrase[1].head = Preposition_Phrase[0]
113         Preposition_Phrase[2].head = Preposition_Phrase[1]
114         Preposition_Phrase[3].head = Preposition_Phrase[2]
115         Preposition_Phrase[4].head = Preposition_Phrase[3]
116
117         pattern_6 = [
118             Token("NN", "NN", None),
119             Token("IN", "IN", "advmod"),
120             Token("IN", "IN", "prep"),
121             Token("NN", "NN", "pobj"),
122             Token("IN", "IN", "prep")
123         ]
124
125         pattern_6[1].head = pattern_6[0]
126         pattern_6[2].head = pattern_6[1]
127         pattern_6[3].head = pattern_6[2]
128         pattern_6[4].head = pattern_6[0]
129
130         Preposition_Phrase_Modifier = [
131             Token("VB", "VB", None),
132             Token("NN", "NN", "dobj"),
133             Token("IN", "IN", "prep"),
134             Token("NN", "NN", "pobj")
135         ]
136         Preposition_Phrase_Modifier[1].head =
    Preposition_Phrase_Modifier[0]
137         Preposition_Phrase_Modifier[2].head =
    Preposition_Phrase_Modifier[1]
138         Preposition_Phrase_Modifier[3].head =
    Preposition_Phrase_Modifier[2]
139
```

```
140         Sub_attach = [
141             Token("PRN", "PRN", "nsubj"),
142             Token("VB", "VB", None),
143             Token("PRN", "PRN", "nsubj"),
144             Token("IN", "IN", "mark"),
145             Token("VB", "VB", "ccomp"),
146             Token("NN", "NN", "npadvmod")
147         ]
148         Sub_attach[0].head = Sub_attach[1]
149         Sub_attach[2].head = Sub_attach[4]
150         Sub_attach[3].head = Sub_attach[4]
151         Sub_attach[4].head = Sub_attach[1]
152         Sub_attach[5].head = Sub_attach[4]
153 
154         Adv_Pos_1 = [
155             Token("VB", "VB", None),
156             Token("RB", "RB", "advmod"),
157             Token("CC", "CC", "cc"),
158             Token("RB", "RB", "advmod"),
159             Token("VB", "VB", "conj")
160         ]
161         Adv_Pos_1[1].head = Adv_Pos_1[0]
162         Adv_Pos_1[2].head = Adv_Pos_1[0]
163         Adv_Pos_1[3].head = Adv_Pos_1[4]
164         Adv_Pos_1[4].head = Adv_Pos_1[0]
165 
166         Adv_Pos_2 = [
167             Token("VB", "VB", None),
168             Token("RB", "RB", "advmod"),
169             Token("VB", "VB", "amod"),
170             Token("NN", "NN", "dobj")
171         ]
172         Adv_Pos_2[1].head = Adv_Pos_2[2]
173         Adv_Pos_2[2].head = Adv_Pos_2[3]
174         Adv_Pos_2[3].head = Adv_Pos_2[0]
175 
176         PP_Adj = [
177             Token("VB", "VB", "aux"),
178             Token("VBG", "VBG", None),
179             Token("NN", "NN", "dobj")
180         ]
181         PP_Adj[0].head = PP_Adj[1]
182         PP_Adj[2].head = PP_Adj[1]
183 
184         graph_patterns_checker = GraphPatternsChecker()
185         graph_patterns_checker.insert_patterns("pattern1",
    pattern_1)
186         graph_patterns_checker.insert_patterns("pattern2",
    pattern_2)
187         graph_patterns_checker.insert_patterns("pattern3",
    pattern_3)
188         graph_patterns_checker.insert_patterns("pattern4",
    pattern_4)
189         graph_patterns_checker.insert_patterns("pattern5",
```

```
    pattern_5)
190         graph_patterns_checker.insert_patterns("ClassAttribute",
    class_attr)
191         graph_patterns_checker.insert_patterns("
    ClassSpecialization", class_special)
192         graph_patterns_checker.insert_patterns("Attributepattern1
    ", Attr_1)
193         graph_patterns_checker.insert_patterns("Attributepattern2
    ", Attr_2)
194         graph_patterns_checker.insert_patterns("
    Generalrelationpattern", General_relation)
195         graph_patterns_checker.insert_patterns("PrepositionPhrase
    ", Preposition_Phrase)
196         graph_patterns_checker.insert_patterns("pattern6",
    pattern_6)
197         graph_patterns_checker.insert_patterns("
    PrepositionPhraseModifier", Preposition_Phrase_Modifier)
198         graph_patterns_checker.insert_patterns("
    SubsentenceAttachment", Sub_attach)
199         graph_patterns_checker.insert_patterns("
    AdverbialPosition1", Adv_Pos_1)
200         graph_patterns_checker.insert_patterns("
    AdverbialPosition2", Adv_Pos_2)
201         graph_patterns_checker.insert_patterns("
    PresentparticiplevsAdjective", PP_Adj)
```

B.3 tests.py

```
1  from graph_patterns_checker import Token, GraphPatternsChecker
2
3
4  def test_sentence1():
5      """ A test function to pass a sentence and check whether it
       returns valid matched patterns or not"""
6      sentence = [
7          Token("Each", "DT", "det"),
8          Token("confirmation", "NN", "compound"),
9          Token("window", "NN", "nsubj"),
10         Token("has", "AUX", None),
11         Token("a", "DT", "det"),
12         Token("close", "JJ", "amod"),
13         Token("button", "NN", "dobj"),
14         Token(".", ".", "punct")
15
16     ]
17
18     sentence[0].head = sentence[2]
19     sentence[1].head = sentence[2]
20     sentence[2].head = sentence[3]
21     sentence[4].head = sentence[6]
22     sentence[5].head = sentence[6]
23     sentence[6].head = sentence[3]
```

```
24      sentence[7].head = sentence[3]
25
26      graph_patterns_checker = GraphPatternsChecker()
27      matched_patterns = graph_patterns_checker.
        get_matched_patterns(sentence)
28
29      assert frozenset(matched_patterns) == frozenset(['pattern1',
        'pattern3', 'pattern4', 'Query', 'ClassAttribute'])
30
31
32  def test_sentence2():
33      sentence = [
34          Token("The", "DT", "det"),
35          Token("rentable", "JJ", "amod"),
36          Token("space", "NN", "nsubj"),
37          Token("is", "VB", "cop"),
38          Token("either", "CC", "cc:preconj"),
39          Token("a", "DT", "det"),
40          Token("hotel", "NN", "compound"),
41          Token("bed", "NN", None),
42          Token("or", "CC", "cc"),
43          Token("a", "DT", "det"),
44          Token("meeting", "NN", "compound"),
45          Token("room", "NN", "conj:or"),
46          Token(".", ".", "punct")
47      ]
48
49      sentence[0].head = sentence[2]
50      sentence[1].head = sentence[2]
51      sentence[2].head = sentence[7]
52      sentence[3].head = sentence[7]
53      sentence[4].head = sentence[7]
54      sentence[5].head = sentence[7]
55      sentence[6].head = sentence[7]
56      sentence[8].head = sentence[6]
57      sentence[9].head = sentence[11]
58      sentence[10].head = sentence[11]
59      sentence[11].head = sentence[7]
60      sentence[12].head = sentence[7]
61
62      graph_patterns_checker = GraphPatternsChecker()
63      matched_patterns = graph_patterns_checker.
        get_matched_patterns(sentence)
64
65      assert frozenset(matched_patterns) == frozenset(['pattern3',
        'pattern4', 'Query', 'ClassAttribute'])
66
67
68  def test_sentence3():
69      sentence = [
70          Token("On", "IN", "case"),
71          Token("the", "DT", "det"),
72          Token("ground", "NN", "compound"),
73          Token("floor", "NN", "nmodon"),
```

```
74          Token(",", ",", "punct"),
75          Token("there", "EX", "expl"),
76          Token("is", "VB", None),
77          Token("a", "DT", "det"),
78          Token("living", "NN", "compound"),
79          Token("room", "NN", "nsubj"),
80          Token(".", ".", "punct")
81      ]
82
83      sentence[0].head = sentence[3]
84      sentence[1].head = sentence[3]
85      sentence[2].head = sentence[3]
86      sentence[3].head = sentence[6]
87      sentence[4].head = sentence[6]
88      sentence[5].head = sentence[6]
89      sentence[7].head = sentence[9]
90      sentence[8].head = sentence[9]
91      sentence[9].head = sentence[6]
92      sentence[10].head = sentence[6]
93
94      graph_patterns_checker = GraphPatternsChecker()
95      matched_patterns = graph_patterns_checker.
        get_matched_patterns(sentence)
96
97      assert frozenset(matched_patterns) == frozenset(['Query', '
        ClassAttribute'])
98
99
100 def test_sentence4():
101     sentence = [
102         Token("The", "DT", "det"),
103         Token("rentable", "JJ", "amod"),
104         Token("space", "NN", "nsubj"),
105         Token("always", "RB", "advmod"),
106         Token("has", "VB", None),
107         Token("a", "DT", "det"),
108         Token("specified", "VBN", "amod"),
109         Token("rent", "NN", "compound"),
110         Token("cost", "NN", "dobj"),
111         Token("and", "CC", "cc"),
112         Token("area", "NN", "dobj"),
113         Token("(", "(", "punct"),
114         Token("measured", "VBN", "dep"),
115         Token("in", "IN", "case"),
116         Token("square", "JJ", "amod"),
117         Token("meters", "NNS", None),
118         Token(")", ")", "punct"),
119         Token(".", ".", "punct")
120     ]
121
122     sentence[0].head = sentence[2]
123     sentence[1].head = sentence[2]
124     sentence[2].head = sentence[4]
125     sentence[3].head = sentence[4]
```

```
126     sentence[5].head = sentence[8]
127     sentence[6].head = sentence[8]
128     sentence[7].head = sentence[8]
129     sentence[8].head = sentence[4]
130     sentence[9].head = sentence[8]
131     sentence[10].head = sentence[8]
132     sentence[10].head = sentence[4]
133     sentence[11].head = sentence[12]
134     sentence[12].head = sentence[10]
135     sentence[13].head = sentence[15]
136     sentence[14].head = sentence[15]
137     sentence[16].head = sentence[12]
138     sentence[17].head = sentence[4]
139
140     graph_patterns_checker = GraphPatternsChecker()
141     matched_patterns = graph_patterns_checker.
        get_matched_patterns(sentence)
142
143     assert frozenset(matched_patterns) == frozenset(
144         ['pattern2', 'pattern3', 'pattern4', 'pattern5', 'Query',
        'Attributepattern1', 'ClassAttribute'])
145
146
147 def test_sentence5():
148     sentence = [
149         Token("He", "PRP", "nsubj"),
150         Token("saw", "VB", None),
151         Token("the", "DT", "det"),
152         Token("man", "NN", "dobj"),
153         Token("with", "IN", "prep"),
154         Token("field", "NN", "compound"),
155         Token("glass", "NN", "pobj"),
156         Token(".", ".", "punct")
157
158     ]
159     sentence[0].head = sentence[1]
160     sentence[2].head = sentence[3]
161     sentence[3].head = sentence[1]
162     sentence[4].head = sentence[3]
163     sentence[5].head = sentence[6]
164     sentence[6].head = sentence[4]
165
166     graph_patterns_checker = GraphPatternsChecker()
167     matched_patterns = graph_patterns_checker.
        get_matched_patterns(sentence)
168
169     assert frozenset(matched_patterns) == frozenset(['Query', '
        PrepositionPhraseModifier'])
170
171
172 def test_sentence6():
173     sentence = [
174         Token("The", "DT", "det"),
175         Token("door", "NN", "nsubj"),
```

```
176         Token("near", "IN", "advmod"),
177         Token("to", "IN", "prep"),
178         Token("stairs", "NN", "pobj"),
179         Token("with", "IN", "prep"),
180         Token("the", "DT", "det"),
181         Token("Members", "NN", "pobj"),
182         Token("Only", "RB", "advmod"),
183         Token("sign", "VB", None),
184         Token(".", ".", "punct"),
185         Token(".", ".", "punct"),
186         Token(".", ".", "punct")
187
188     ]
189
190     sentence[0].head = sentence[1]
191     sentence[1].head = sentence[9]
192     sentence[2].head = sentence[1]
193     sentence[3].head = sentence[2]
194     sentence[4].head = sentence[3]
195     sentence[5].head = sentence[4]
196     sentence[6].head = sentence[7]
197     sentence[7].head = sentence[5]
198     sentence[8].head = sentence[9]
199
200     graph_patterns_checker = GraphPatternsChecker()
201     matched_patterns = graph_patterns_checker.
    get_matched_patterns(sentence)
202
203     assert frozenset(matched_patterns) == frozenset(['Query', '
    pattern6', 'PrepositionPhrase'])
204
205
206 def test_sentence7():
207     sentence = [
208         Token("He", "PRN", "nsubj"),
209         Token("said", "VB", None),
210         Token("that", "IN", "mark"),
211         Token("she", "PRN", "nsubj"),
212         Token("had", "VB", "aux"),
213         Token("done", "VB", "ccomp"),
214         Token("it", "PRN", "dobj"),
215         Token("yesterday", "NN", "npadvmod")
216     ]
217
218     sentence[0].head = sentence[1]
219     sentence[2].head = sentence[5]
220     sentence[3].head = sentence[5]
221     sentence[4].head = sentence[5]
222     sentence[5].head = sentence[1]
223     sentence[6].head = sentence[5]
224     sentence[7].head = sentence[5]
225
226     graph_patterns_checker = GraphPatternsChecker()
227     matched_patterns = graph_patterns_checker.
```

```
    get_matched_patterns(sentence)
228
229     assert frozenset(matched_patterns) == frozenset(['Query', '
    SubsentenceAttachment'])
230
231
232 def test_sentence_8():
233     sentence = [
234         Token("The", "DT", "det"),
235         Token("lady", "NN", None),
236         Token("you", "PRN", "nsubj"),
237         Token("met", "VB", "relcl"),
238         Token("now", "RB", "advmod"),
239         Token("and", "CC", "cc"),
240         Token("then", "RB", "advmod"),
241         Token("came", "VB", "conj"),
242         Token("to", "TO", "aux"),
243         Token("visit", "VB", "advcl"),
244         Token("us", "PRN", "dobj"),
245         Token(".", ".", "punct")
246
247     ]
248
249     sentence[0].head = sentence[1]
250     sentence[2].head = sentence[3]
251     sentence[3].head = sentence[1]
252     sentence[4].head = sentence[3]
253     sentence[5].head = sentence[3]
254     sentence[6].head = sentence[7]
255     sentence[7].head = sentence[3]
256     sentence[8].head = sentence[9]
257     sentence[9].head = sentence[7]
258     sentence[10].head = sentence[9]
259
260     graph_patterns_checker = GraphPatternsChecker()
261     matched_patterns = graph_patterns_checker.
    get_matched_patterns(sentence)
262
263     assert frozenset(matched_patterns) == frozenset(['Query', '
    AdverbialPosition1'])
264
265
266 def test_sentence9():
267     sentence = [
268         Token("A", "DT", "det"),
269         Token("secretary", "NN", "nsubj"),
270         Token("can", "VB", "aux"),
271         Token("type", "VB", None),
272         Token("quickly", "RB", "advmod"),
273         Token("written", "VB", "amod"),
274         Token("reports", "NN", "dobj")
275     ]
276     sentence[0].head = sentence[1]
277     sentence[1].head = sentence[3]
```

```
278     sentence[2].head = sentence[3]
279     sentence[4].head = sentence[5]
280     sentence[5].head = sentence[6]
281     sentence[6].head = sentence[3]
282 
283     graph_patterns_checker = GraphPatternsChecker()
284     matched_patterns = graph_patterns_checker.
        get_matched_patterns(sentence)
285 
286     assert frozenset(matched_patterns) == frozenset(['Query', '
        Attributepattern1', 'pattern2', 'AdverbialPosition2'])
287 
288 
289 def test_sentence10():
290     sentence = [
291         Token("We", "PRN", "nsubj"),
292         Token("are", "VB", "aux"),
293         Token("writing", "VBG", None),
294         Token("a", "DT", "det"),
295         Token("letter", "NN", "dobj")
296 
297     ]
298 
299     sentence[0].head = sentence[2]
300     sentence[1].head = sentence[2]
301     sentence[3].head = sentence[4]
302     sentence[4].head = sentence[2]
303 
304     graph_patterns_checker = GraphPatternsChecker()
305     matched_patterns = graph_patterns_checker.
        get_matched_patterns(sentence)
306 
307     assert frozenset(matched_patterns) == frozenset(['Query', '
        PresentparticiplevsAdjective'])
```

Appendix C

Contents of enclosed CD

- `readme.txt` ..... the file describes the contents of the CD
- `src` ..... the directory of source codes
  - `implementation` ..... implementation source codes
  - `thesis` ..... source codes of the thesis in LaTeX
- `text` ..... the directory of thesis text document
  - `DP_Manoharan_Vigneshwar_2022.pdf` .... the thesis in a PDF format