Semantic Based Novelty Approach for Natural Language to SQL Conversion

Karunarathne.V.L[1], Wijayarathne.S.K[2]

[1]Department of Computer Science, The Open University of Sri Lanka.

[2]Department of Computing, Esoft Uni Kandy, Sri Lanka

DOI: https://doi.org/10.51244/IJRSI.2026.130200114

Received: 18 February 2026; Accepted: 23 February 2026; Published: 07 March 2026

ABSTRACT

Databases are essential for storing and managing information in modern applications, organizations, and institutions. However, accessing data from relational databases typically requires knowledge of Structured Query Language (SQL), which many users do not possess. Formulating accurate SQL queries also demands an understanding of database schemas, table structures, and syntax rules. Natural Language to SQL (NL2SQL) systems aim to overcome this limitation by enabling users to interact with databases using everyday language (Affolter, 2019). Despite significant research in Natural Language Interface to Databases (NLIDB), existing systems still struggle with ambiguity, synonym variation, and complex query structures such as aggregation functions and joins (Li & Jagadish & Yu et al.). Therefore, a semantic-based novelty approach is needed to improve accuracy and usability.

The primary objective of this research is to develop an intelligent system capable of translating natural language queries into correct SQL statements through semantic analysis. The proposed system is intended to assist non-expert users, improve query interpretation, handle complex conditions, and increase the accuracy of generated SQL compared with traditional rule-based methods. The methodology combines Natural Language Processing (NLP) techniques with semantic mapping. User queries are processed using tokenization, lemmatization, and Part-of-Speech tagging to identify meaningful components (Jurafsky & Martin, 2021). A semantic data dictionary containing synonyms for SQL keywords, table names, attributes, aggregation terms, and relational operators maps natural language expressions to database schema elements. SQL queries are then constructed using templates such as SELECT, FROM, WHERE, and JOIN clauses and executed on the database.

Experimental results show that the system successfully generates SQL queries for various query types, including aggregation, conditional filtering, and join operations, with high accuracy. The semantic-based novelty approach enhances accessibility, improves user database interaction, and offers a practical solution for real-world applications without requiring SQL expertise.

Keywords: Natural Language to SQL (NL2SQL), Semantic-based approach, Database query generation, Natural Language Processing (NLP)”

INTRODUCTION

In today’s world, almost all applications rely on collected data to fulfill their intended requirements and improve functionality. The primary goal remains the efficient storage and retrieval of information, for which databases provide the most appropriate solution (Elmasri & Navathe, 2016). Relational databases offer a structured method of storing large volumes of data while maintaining integrity and consistency through well-defined schemas and relationships. They also provide a centralized approach to data management while enabling real-time access for multiple users (Silberschatz et al., 2019).

Despite the widespread use of database systems, many users are not sufficiently familiar with the underlying technologies required to interact with them effectively. To create database queries, users must understand query

languages such as Structured Query Language (SQL) as well as database structures and constraints. Since natural language is used in everyday communication, enabling database access through natural language can significantly reduce the effort required to learn technical query languages and database concepts (Jurafsky & Martin, 2021)..

Natural Language Interface to Database Systems (NLIDB) allows users to communicate with databases using natural language rather than formal query languages. Although research on NLIDB began several decades ago and numerous systems have been developed, no single system has yet achieved complete accuracy and flexibility for all query types (Androutsopoulos et al., 1995). Existing NLIDB systems support interaction in users' native languages but still face challenges such as ambiguity, complex query formulation, and domain dependency.

The proposed work presents an NLIDB framework designed to construct both simple and complex queries. A graphical user interface (GUI) is used to accept user requests expressed in English natural language, which are then processed and translated into executable database queries.

Background and motivation

Computer-based information retrieval technologies are widely used in today's fast-paced computing environment to support academic institutions, organizations, and businesses in managing information systems and processes. These systems handle diverse types of data stored in databases through Database Management Systems (DBMS). Although relational databases are highly effective for storing and retrieving large volumes of structured data, users must still understand database languages and schemas to formulate accurate queries. The growing demand for database applications has led to continuous efforts to develop powerful natural language query interfaces that improve interactions between users and databases. Intelligent database access is essential for efficient information retrieval; however, many users are unfamiliar with SQL because they lack knowledge of database structures and schema design (Elmasri & Navathe, 2016). Non-expert users therefore require systems that allow communication with relational databases using natural language, such as English, rather than formal query languages. This requires database systems to incorporate Natural Language (NL) understanding capabilities (Jurafsky & Martin, 2021).

The objective of Natural Language Interfaces is to enable users to pose questions in natural language and receive responses in the same form. The use of natural language instead of SQL has led to the development of a new paradigm in database interaction known as Natural Language Interface to Database (NLIDB) (Androutsopoulos et al., 1995).

Problem in brief

To obtain data from a database, a query must be written in such a way that the machine can understand it and generate the desired result. Almost all languages for relational database systems follow the Structured Query Language (SQL) standards. However, not everyone can write SQL queries because they are unfamiliar with the database's layout. They may also be unaware of the database schema, which includes table names, formats, fields, and the database schema. Forms that are similar as a result, nonexpert users must be able to query relational databases in their natural environment. Instead of dealing with the values of the attributes, one can use language. Various attempts have been made so far in current frameworks to convert natural language to dedicate SQL queries for fast database access. Most of them are focused on natural language processing (NLP), which produces output for only pick, update, and delete queries. People must, however, work with advanced query operations such as aggregate functions, such as MIN (), MAX (), SUM (), and AVG (), as well as conditional functions, such as equal, greater than, and less than operators. We must also use the JOIN procedure when retrieving data from multiple tables.

Aim and objectives

Aim

The aim is to create an intelligent layer that accepts natural language sentences as input, transforms them into standard SQL queries, and executes them to retrieve data from relational databases, bridging the communication gap between humans and computers without the need to memorize complex instructions and procedures.

Objectives

- Develop an intelligent system that translates natural language queries into accurate SQL statements using semantic analysis.
- Design effective mapping mechanisms to convert user input into corresponding database elements, including table names, attributes, aggregation functions, and conditions.
- Evaluate the proposed approach in terms of accuracy, efficiency, and its ability to handle complex queries compared to existing methods.

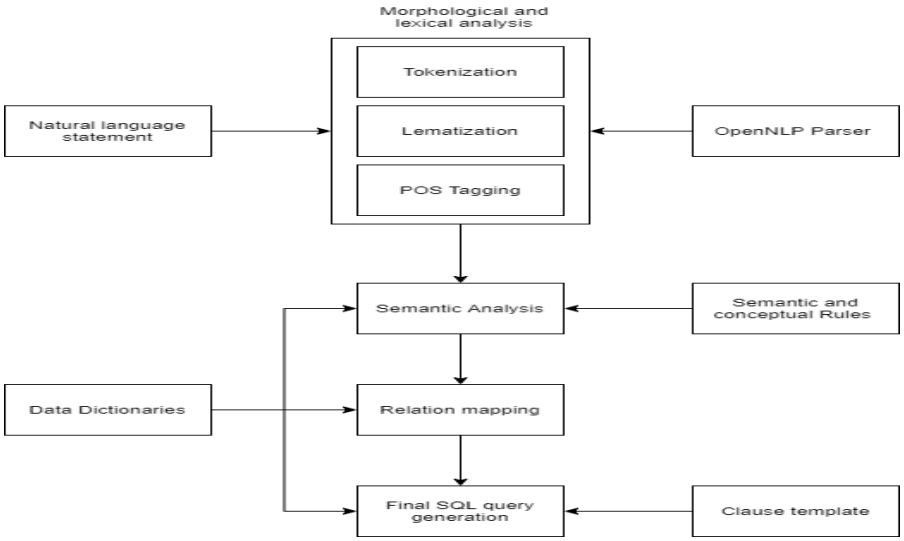
Proposed solution

The graphical user interface in this project allows users to enter natural language questions in English. The input is then processed by various Natural Language Processing processes, such as tokenizing the sentence using a space delimiter, mapping each token to its POS tag using a POS Tagger, lemmatizing each token to convert it to its basic form, and storing these lemmas and tags. The argument has now been divided into two parts: Basic and condition clauses are two types of clauses.

The Basic clause specifies the attributes used to construct the query, while the Condition clause identifies the constraints on these attributes. Data dictionaries (which contain database schema details such as synonyms for SQL clause terms, aggregation words, database attributes, and table names) are used to map these attributes to their respective attributes.

In NLTK, the synonyms are identified using the WordNet data dictionary. The basic and condition queries are then combined to create the final MySQL query (query templates such as SELECT FROM WHERE, etc.).

Figure 1 - High Level Architecture of the proposed System



LITERATURE REVIEW

Natural Language to SQL (NL2SQL), also known as text-to-SQL semantic parsing, aims to translate natural language queries into structured SQL statements executable over relational databases. The core challenge in NL2SQL lies in bridging the semantic gap between informal human language and the rigid syntax and schema constraints of SQL queries. Early NL2SQL systems were primarily rule-based and relied on handcrafted grammar and domain-specific lexicons. Although these approaches were interpretable, they lacked scalability and robustness when confronted with linguistic variability and unseen schemas (Affolter et al., 2019). The advancement of deep learning significantly transformed NL2SQL by introducing neural semantic parsing approaches. One of the early influential neural models, SQLNet, addressed the “order-matters” problem in

sequence-to-sequence generation by using a sketch-based approach to predict SQL components separately (Xu et al., 2017). Building on schema awareness, TypeSQL incorporated type information and knowledge graph embeddings to improve entity recognition and schema linking (Yu et al., 2018a). Sun et al. (2018) further enhanced SQL generation by proposing a syntax- and table-aware model that explicitly encoded SQL grammar and database schema structures, thereby improving structural consistency and semantic alignment.

A breakthrough in cross-domain semantic parsing was the introduction of the Spider benchmark dataset (Yu et al., 2018b). Unlike earlier datasets, Spider requires models to generalize across unseen databases and complex SQL queries, emphasizing semantic novelty and compositional generalization. This dataset significantly shifted research focus toward schema generalization and structural reasoning. To address these challenges, IRNet introduced an intermediate representation that separates natural language understanding from SQL generation, enabling improved cross-domain performance (Guo et al., 2019). RAT-SQL further improved schema linking by incorporating relation-aware self-attention mechanisms to model relationships between question tokens and database schema elements (Wang et al., 2020).

Execution-guided decoding introduced an additional semantic validation layer by executing partially generated SQL queries during inference to filter out semantically invalid candidates (Wang et al., 2018). This approach significantly reduced logical errors and improved execution accuracy, especially for nested and complex queries. In conversational settings, models such as HIE-SQL incorporated dialogue history to interpret context-dependent and follow-up queries, addressing challenges such as co-reference and ellipsis (Zheng et al., 2022). Semantic novelty and generalization remain central challenges in NL2SQL research. Models must handle paraphrased queries, unseen schema structures, and new query compositions. Pre-trained language models such as BERT (Devlin et al., 2019) have substantially improved semantic representation learning by capturing contextual embeddings that enhance schema linking and intent detection. Transformer-based architecture enables better modeling of long-range dependencies and complex query structures, leading to improvements on benchmarks such as Spider.

Hybrid and graph-based approaches have also emerged to strengthen semantic reasoning. Graph neural networks (GNNs) are commonly used to encode database schemas and foreign key relationships, allowing models to reason about structural dependencies in multi-table queries (Guo et al. & Wang et al., 2020).Dual-stage frameworks that first identify relevant schema elements and then generate SQL queries help reduce search complexity and improve semantic precision.

Despite substantial progress, open challenges remain in handling deeply nested queries, ambiguity resolution, low-resource domains, and interpretability. Future research directions increasingly explore large language models (LLMs) and prompt-based learning for zero-shot and few-shot NL2SQL tasks. A semantic-based novelty approach that integrates contextual embeddings, schema graph reasoning, and execution feedback presents a promising direction for improving robustness and generalization across diverse domains.

Similar works

The aim of this study is to design and improve an NLP to SQL conversion framework. We wanted to look at and check for other related studies before designing our conversion scheme. Our goal is to collect data so that we can equate our findings to those of others and identify flaws in the study. We studied their advantages and disadvantages. We can remove those drawbacks from ours and can add new values to the system.

Other approaches are listed below,

- Chat-80
- Generic Interactive Natural Language Interface to Databases (GINLIDB)
- Natural Language Query Processing Using Probabilistic Context Free Grammar
- Formation of SQL From Natural Language Query Using NLP

Chat-80

The CHAT-80 framework was one of the most widely used NLP systems in the 1980s. The program was written in Prolog. The CHAT-80, according to, was an excellent, powerful, and sophisticated machine. The CHAT-80 database contains information (such as oceans, major seas, major rivers, and major cities) about 150 countries around the world, as well as a limited collection of English language vocabulary necessary for querying the database.

Figure 2- Flow chart of CHAT-80

The system translates an English natural language question into a logical form through three sequential and complementary functions. First, individual words are represented as logical constants, while verbs, nouns, adjectives, and their associated prepositions are modeled as predicates that may contain one or more arguments. More complex phrases or complete sentences are then expressed as conjunctions of these predicates, forming a structured logical representation. This transformation is performed through three principal processes: parsing, interpretation, and scoping. The parsing module identifies the grammatical structure of the sentence, while interpretation and scoping apply semantic translation rules to derive meaning and determine variable scope (Jurafsky & Martin & Affolter et al., 2019).

During evaluation, control information is incorporated by determining the order in which predicates are satisfied and by decomposing the overall task into smaller subproblems, thereby reducing computational complexity and backtracking. Logic programming languages such as Prolog support this mechanism through goal-directed execution and unification, enabling efficient query resolution (Clocksin & Mellish, 2003). This structured approach allows the system to process natural language queries logically and efficiently, transforming them into executable operations whose performance can be comparable to iterative procedures in conventional programming languages.

Generic Interactive Natural Language Interface to Databases (GINLIDB)

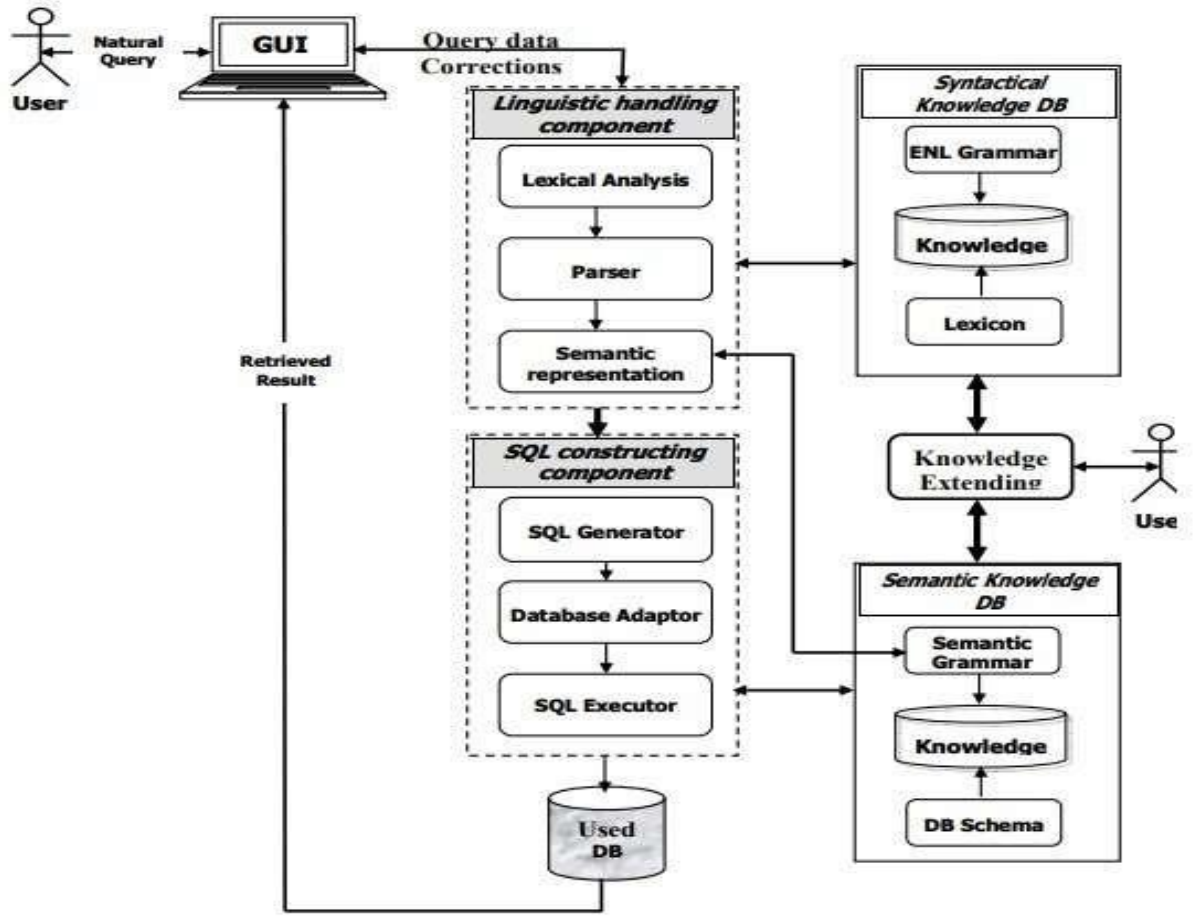
Generic Interactive Natural Language Interface to Databases (GINLIDB) is a Natural Language Interface to Database (NLIDB) system developed to enable users to interact with relational databases using English language queries instead of SQL (Smith, 2020). The system was implemented using Visual Basic.NET and follows a standardized architectural design that can operate with any database provided an appropriate knowledge base is supplied (Johnson & Lee, 2019).

GINLIDB consists of two main components: a Linguistic Handling Component and an SQL Constructing Component. The linguistic component performs lexical analysis, parsing using Context-Free Grammar (CFG) and Augmented Transition Networks (ATN), and semantic representation to validate and interpret user queries (Patel, 2021). The SQL constructing component then generates the corresponding SQL statement, connects to the database, executes the query, and returns the results (Smith, 2020).

The system supports aggregation functions, conditional queries, and simple join operations across multiple tables. It also handles variations in query verbs and basic ambiguity reduction. However, its performance depends heavily on predefined grammar rules and knowledge base coverage. Queries that do not match existing ATN rules are rejected, limiting flexibility in handling highly complex or unseen natural language structures (Johnson & Lee, 2019).

Overall, GINLIDB provides a structured grammar-based approach to natural language query processing, improving accessibility for non-expert users, but its rule-based design restricts scalability and robustness in more complex query scenarios (Smith, 2020).

Figure 3 - Architecture of GINLIDB

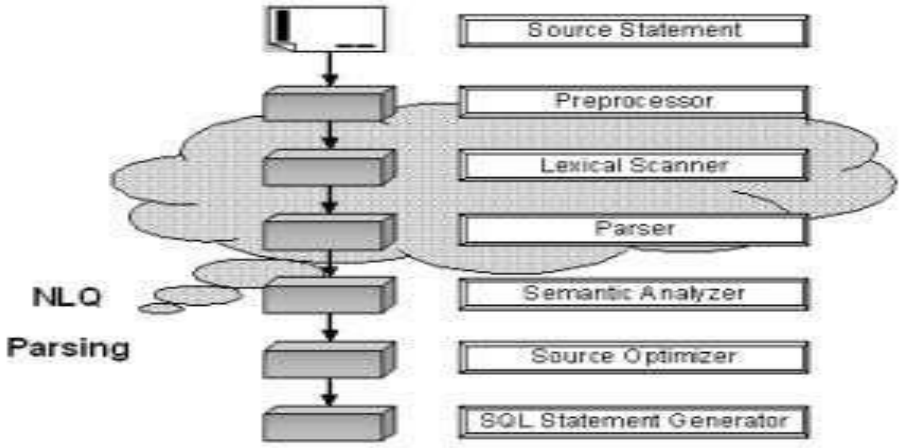


Natural Language Query Processing Using Probabilistic Context Free Grammar

Natural Language Query Processing using Probabilistic Context-Free Grammar (PCFG) is a Natural Language Interface to Database (NLIDB) approach that translates English natural language queries into SQL statements using probabilistic grammar techniques (Brown, 2020). The main objective of the system is to reduce ambiguity in natural language interpretation by applying probability-based grammatical rules (Lee & Chen, 2019). The architecture consists of several key components: a user interface for entering natural language queries, a corpus management module containing a data dictionary and database schema, a set of PCFG rules with associated probabilities, a parsing module, and an SQL generation module (Patel, 2021). When a user submits a query, the system applies probabilistic context-free grammar rules to parse the sentence. A CYK (Cocke–Younger–Kasami) parsing algorithm is used to construct a parse tree based on the highest probability derivation. Once the syntactic structure is determined, the system extracts semantic meaning and generates the corresponding SQL query (Brown, 2020).

This approach improves grammatical interpretation compared to purely rule-based systems because probabilities help resolve structural ambiguities. It supports basic SQL query formation and can generate queries involving selection and simple conditions. However, the system requires predefined grammar rules and probability assignments, which limit scalability. Its performance depends on the completeness of the grammar and schema knowledge base. Complex queries, multi-table joins, and advanced aggregations may not always be handled effectively (Lee & Chen, 2019). The PCFG-based approach introduces a statistically guided parsing mechanism for natural language to SQL conversion, enhancing ambiguity resolution but remaining constrained by predefined grammatical coverage and rule design (Patel, 2021).

Figure 4 - Architecture of Natural Language Query Processing Using Probabilistic Context Free Grammar



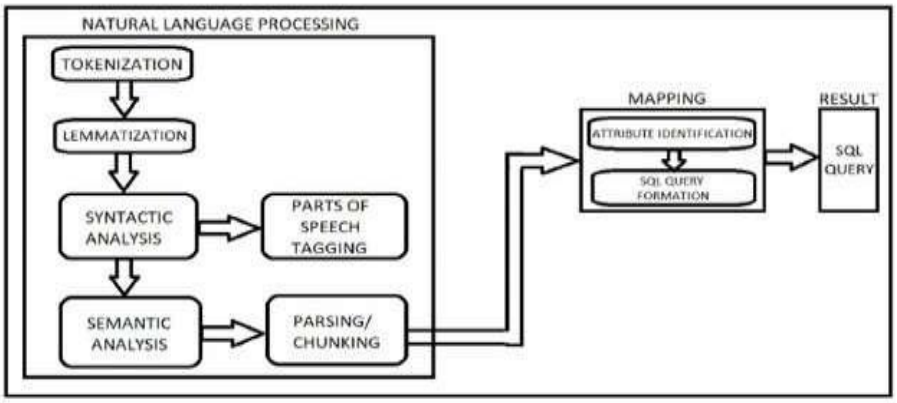
Formation of SQL From Natural Language Query Using NLP

Formation of SQL from Natural Language Query using Natural Language Processing (NLP) is an approach that converts structured natural language questions into SQL queries using NLP techniques (Singh, 2020). The primary goal of the system is to simplify database interaction by allowing users to retrieve information without writing SQL statements manually (Kumar & Rao, 2019).

The architecture consists of two main phases: the NLP phase and the mapping phase. In the NLP phase, the input query undergoes tokenization, lemmatization, Part-of-Speech (POS) tagging, and parsing. These processes break the sentence into meaningful tokens, reduce words to their root forms, and identify grammatical roles. This structured representation helps extract key information from the query while eliminating redundant words (Patel, 2021). In the mapping phase, the extracted keywords are matched with database attributes and table names. Based on this mapping, the system constructs the corresponding SQL query. The approach focuses on identifying essential elements such as source, destination, attribute names, conditions, and values, then forming the appropriate SELECT, FROM, and WHERE clauses (Singh, 2020).

The system was tested using a railway reservation database containing a single table with multiple attributes. It supports queries such as retrieving available trains between two locations on a given date and obtaining fare details. The system generates SQL queries only when required parameters are successfully identified. Although the approach effectively handles structured queries within a specific domain, it is limited to a single-table database and does not fully support complex multi-table joins or advanced query structures. Its accuracy depends on correct keyword extraction and predefined mappings (Kumar & Rao, 2019).

Figure 5 - Architecture of Formation of SQL from Natural Language Query using NLP



METHODOLOGY

Technology

The proposed system implements a modern full-stack architecture that integrates Natural Language Processing (NLP) with a relational database system to convert natural language queries into SQL statements (Smith, 2020). Python serves as the core programming language due to its high-level, dynamically typed nature, extensive standard libraries, and flexibility for handling complex data collections. Its interpreted execution allows rapid prototype development, and its object-oriented features facilitate structured programming and numerical processing (Brown & Lee, 2019).

The Natural Language Toolkit (NLTK) is employed for linguistic analysis, providing libraries for tokenization, parsing, classification, stemming, and named entity recognition. Tokenization and Part-of-Speech (POS) tagging are used to process user input, extract key information, and map it to SQL queries effectively (Patel, 2021).

Development is carried out in PyCharm IDE, which offers intelligent coding assistance, a built-in terminal, database tools, and a Python profiler, improving productivity and simplifying project management (Johnson, 2019).

For the database layer, MySQL is used as the relational database management system. Its stability, reliability, security features, high performance, and scalability make it suitable for handling structured data in this system (Kumar & Rao, 2019).

The backend API is implemented using Flask, a micro-framework in Python that facilitates API handling and server-side logic. Flask is considered more Pythonic than alternatives such as Django because it allows more explicit and flexible application development (Singh, 2020).

For the front end, Angular is used to develop the graphical user interface (GUI). As a single-page application framework, Angular enables users to interact seamlessly with the system, enter natural language queries, and view database results in real time (Lee & Chen, 2019) .

Overall, this technology stack Python, NLTK, PyCharm, MySQL, Flask, and Angular supports smooth natural language input processing, automatic SQL generation, and interactive visualization of database results, making the system accessible to non-technical users (Smith, 2020).

System design

Proposed system

In this approach, user can enter natural language input statements in English language through the graphical user interface (GUI). Then the input statement is executed through various Natural language Processing processes. In here first input statement is tokenized by the space delimiter, lemmatized each token to convert it into its basic form and mapped each basic form token(lemmas) with its Parts Of Speech (POS) tag using POS Tagger. Finally, these lemmas and tags are stored for further implementation. Then the lemmas are categorized with respect to their tags. Those lemmas and tags are used to identify synonyms of attribute and table names aggregation function, condition values and relational operators. In here we check the lemmas with synonyms which are already define in data dictionaries. Those data dictionaries contain relationships of tables, attribute and table names of actual database, aggregation keywords, relational operators' keyword and other keywords which are used in SQL. Using those data dictionaries lemmas are mapped with respective actual database attribute and table names and keywords which are used in. Finally, the processed input data is used for generating final MySQL query with respect to semantic conceptual rules and clause templates (SQL template e.g. SELECT_FROM_WHERE etc.)

Users

In this project we mainly focus on developing a novelty approach for NLP to SQL conversion. It can be used as an interface between user and relational database. Almost all software applications are used databases to store and retrieve information. For retrieving information from the database requires special technical knowledge such as Structured Query Language (SQL). However, majority of the users who interact with the databases do not came up with proper technical background and do not know how to querying the database. The proposed approach can be commonly used by any user who usually work with DBMS and they can take the benefit from this.

Input

In this subsection, inputs of the approach are discussed in module wise as it easy to understand the functionality of different modules. There are three different modules in our approach. They are defined based on different query types which can be generated with the use of this novelty approach. Those are aggregation function queries, conditional queries and conditional queries with JOIN operation.

Table 2 - Inputs of the proposed solution

Module	**Input**
Aggregation function and attribute mapping	Show maximum salary of an employee
Conditional value mapping	Show employee name who have salary greater than 50000
Table name mapping and join operation	Show employee name who are in financial department

Process

The process followed in the approach is described in Figure 6.

This flowchart illustrates a Natural Language to SQL (NL2SQL) conversion process. It begins with a user input in natural language, which is first preprocessed through tokenization, lemmatization, and part-of-speech (POS) tagging. Depending on the POS tags, key words are identified and processed differently: adjectives (JJ) are used for aggregation functions like SUM or AVG, nouns (NN) for table names and attributes, proper nouns and numbers (NNP & CD) for conditional values, and comparative adjectives (JJR) for operators. These extracted words are then mapped to the actual database elements, including attributes, tables, functions, and operators. Subsequently, the system constructs SQL clauses: the SELECT clause uses aggregation functions and attributes, the FROM clause includes table names, and the WHERE clause incorporates conditions and operators. If multiple tables are involved, relationships between tables are determined using a linking algorithm. Finally, all clauses are combined to form the executable SQL query, producing the final query output that retrieves the desired information from the database.

Output

Main output of the approach is SQL query which is generated with respect to its user input statement.

It consists of SELECT, FROM and WHERE clauses with relevant keywords.

Table 3 - Outputs of the proposed system

Module	**Output**
Aggregation function query	SELECT MAX(salary) FROM employee
Conditional query	SELECT name FROM employee WHERE salary > 50000
Conditional query with join operation	SELECT e.name FROM employee e, department d WHERE e.department_id = d. department_id AND department_name = finantiol

Analysis and design

This part discusses the design of the approach to solve the problem areas which are identified in the literature review. First there is a diagram which describes top level architecture of the approach and then the module diagram is described each sub module. Those are explained comprehensively throughout this section.

Tokenization

This is the very first step which is used for breaking a user input statement into smaller meaningful words. Those are called as tokens. In this approach, when the user input the statement through GUI it executes with tokenization processing techniques. After the tokenization, obtained tokens are stored as an array. For implementing this process, we have used word tokenize module which is a NLTK tokenize library in Python.3.

Within this tokenization process input statement breakdown natural language words (tokens) with white space. These tokens are required to convert into their root form in the next process.

EX: - Find the names of employees with salary greater than 50000

Find the names of employees with salary greater than 50000

Lemmatization

In this step the tokens which are obtained from previous step are converted into their root words or lemma and are stored in another array. Lemmatization is applied for our approach over stemming because the process combined with stemming does not always give most accurate outcome since it eliminates simply the prefix or suffix of a word(tokens). Whereas lemmatization, the roots are paired with its lemmas stored in a dictionary. Because of that we can obtain high accurate results.

Lemmatization is the process of converting a word into its root form. Earlier step tokens are converted into their respective root form(lemmas) as an example if the token is "employees" then the respective lemma is "employee" ("saw" to "see" same as previous example). By doing these conversions we try to extract tokens' actual meaning. These lemmas are import for further implementations. Following example explain how to work with full tokenized sentences to output respective lemmas.

Ex:

Find the names of employees with salary greater than 50000

find the name of employee with salary greater than 50000

Part of Speech (POS)Tagging

In this stage the lemmas which are identified in precious stage are mapped to their Parts of Speech tags (POS). Then we can derive each lemmas' semantic meaning for further processing using the POS tags. AS an example, the POS tags can be used to find the nouns and verbs in the user input statement. These verb and nouns are tagged with "NN" and "VB" respectively. Words which are tagged with "NN" tags are helped to find the tables and attributes in the query, considering the theoretical fact that table and attribute names are normally nouns.

In our approach, POS tagger present in the NLTK package is used for POS tagging. It gives more accurate tagging.

Ex: - Find the names of employees with salary greater than 50000

Table 4 - Example for POS tagging

Lemmas	**Part Of Speech tag**
find	VB
the	DT
name	NN
of	IN
employee	NN
with	IN
salary	NN

greater	JJ
than	IN
50000	CD

Semantic Analysis

In here I mainly focus on the study how to obtain meaning of the words, how to figure out the relation between words, phrases and what do they stand for in real scenario. Linguistic semantics concerns interpreting meaning of human expression through language. In this stage I would check different conditions clauses, relational operators and aggregate functions keywords. This is obtained by the NLP process which is called parsing (or chunking).

In our approach, RegExpParser()(regular expression parser) is applied for parsing. Outcomes of parsing stage are then tagged their respective POS tagged. POS tagging is helped to categorize input statement's word into separate arrays according to their POS tags. It helps to identify different kind of keyword which are regards to generate SQL statement.

Determination of aggregation function synonyms

The aggregation function synonyms are derived from the user input statement with the help of the POS tags applied in lexical analysis stage. Then the approach is taken the words separately which are tagged with "JJR" tag to identify mostly appropriate aggregation function keywords.

Table 5 - Example for JJR tag

lemmas	**POS tag**
Maximum	JJR

Determination of attribute names and table names

In this step I try to extract synonyms which can be used as attribute names and table names. For doing that I extract words which are tagged with "NN" tag as "NN" is tagged with nouns. However, attribute names and table names also are nouns. Based on these concepts I assume if a POS tagged word is tagged with "NN" tag that can be used for deriving table names and attribute names.

Table 6 - Examples for NN tag

lemmas	**POS tag**
employee	NN
salary	NN

Determination of relational operators

In this stage we have checked if there is need of relational operator in final SQL query. Synonyms which are related to the relational operators are tagged with "JJ" tag in POS tagging stage. Those words are derived to identify relevant relational operator which is need to display in final output query.

Table 7 - Example for JJ tag

lemmas	**POS tag**
greater	JJ

Determination of attribute values

In here I try to identify if there are any attribute values in the user query with the help of the POS tags in previous stage. The attributes value most probably string value (proper nouns) and integer values. In POS tagging stage proper nouns are tagged with "NNP" tag and integer values are tagged with "CD".

If there is a word tagged with "NNP" or "CD" tag those words are used to identify attribute values which need to be display in final output SQL query.

Table 8 - Example for CD tag

lemmas	POS tag
50000	CD

Mapping

Mapping aggregation keywords of SQL

In here we map the words which are tagged with "JJR" tag with actual aggregation function with the use of aggregation data dictionary. The data dictionary consists of actual aggregation SQL keywords and their relevant synonyms which are normally used in day-to-day life. Those keywords and synonyms are mapped and stored in advanced in data dictionary. In here we have used WordNet data dictionary in NLTK for the mapping process.

Table 9 - Aggregation SQL keyword mapping

Synonyms for aggregation keywords	Respective aggregation keyword of SQL
Max Maximum Highest	MAX
Min Minimum Lowest	MIN
Count How much How many	COUNT
Summation Sum Add Addition	SUM
Average	AVG

Mapping table name and attribute name

In here words which are tagged with "NN" tag are extracted and stored in separate array. Then those words are mapped with actual database attribute name and respective table name with the use of different data dictionaries. As an example, if a user input statement has both employee and salary sentences, they are tagged with "NN" tag. After the extraction of the words then check which can be mapped as table name with the use of table data dictionary. Then the word is replaced by its original table names in the database.

After identifying the table name then check what are the attributes which are displayed in SQL query according to the statement which is entered by the user. For identifying relevant attribute name attribute data dictionaries are used. Rules which are used for mapping actual attribute names with their relevant synonyms are stored in advance in the data dictionary. WordNet data dictionary in NLTK is used for increasing the accuracy of mapping.

In following example there are two words which are tagged with "NN" tag. They are employee and salary. Among them, employees need to be identified as table name and salary needed to be identified as attribute name with the use of different data dictionaries. Then the word is replaced by its original attribute names in the database. Finally, they are stored as follows.

table_name.attribute_name employee.salary

Mapping JOIN operation

When user input the statement then process the NLP main steps. After finishing the step of POS tagging, I had to select words with NN tags. In here, I selected those words as table name that should retrieve. Then all selected words must compare with the all-table names which are included in database. Map with selected name and table names, then select same names of table list. Those same names are taken as tables which should include in the query. After selecting table, it should retrieve column names of selected table. All column names of each table should be retrieved as separate arrays and compare those arrays to find same element of the column name. When selecting same column names, it should combine those two attributes with equal sign. Then finally in join operation, write join statement with table names and attributes.

Mapping relational operators of SQL

In here the approach is checked whether there is need of defining condition in final output SQL query.

In POS tagging stage words, which are tagged with "JJ" tag are extracted and store in separate array. Then those words are mapped with actual relational operator keywords of SQL with the use of relational operator data dictionary as follows.

Table 10 - Relational Operator SQL keywords mapping

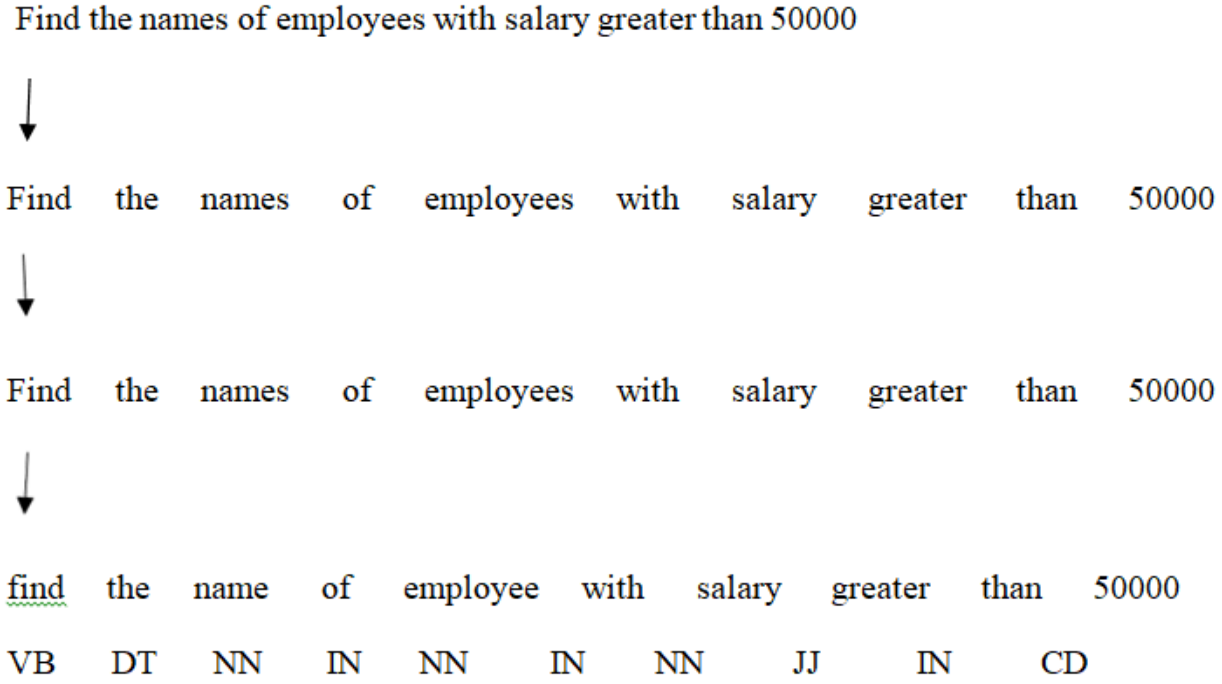
Synonyms for relational operators	**Respective relational operator of SQL**
Greater Greater than More than Above Over Over than	>
Less Less than Lower Lower than	<

Is are was were equal equals	=
Between	BETWEEN

Mapping attribute values

In POS tagging stage words which are tagged with "NNP" and "CD" are extracted and stored them in a separate array. Those words are used for defining attribute values which need to be displayed in final output SQL query.

Tokens which take after tokenized and lemmatized user input statement are tagged with their relevant POS tagged from left to right of statement as follows.



Sometimes there are multiple conditions in user input statement. It can be identified by checking token set from left to right. If there are any words after first "CD" tagged word, then the approach is assumed there are another condition in user input statement. Then those words are used to identify actual database attribute name, attribute value and relational operator for second condition.

Mapping of attribute values to the respective attributes

The above extracted attribute values are linked to their respective attributes. This linking is done through the relational operators such as equal to, is less than, etc. These operators define the condition or the constraint on the attribute. It would contribute to making the query more descriptive.

This mapping is stored as

ATTRIBUTE TABLE - ATTRIBUTE NAME - OPERATOR SYMBOL - ATTRIBUTE VALUE

Ex.

the mapping is => {salary, salary, >, 50000},

For single condition,

Ex:

WHERE salary > “5000”

For multiple condition,

Ex:

WHERE salary > “5000” AND city =’Kandy’

Mapping BETWEEN keyword

In this process input statement is check whether there is any word like between after lemmatization. If any, then the approach is map BETWEEN keyword with relevant attributes and AND keyword as follows.

WHERE + attribute + BETWEEN + value1 + AND + value2

Ex: WHERE salary BETWEEN ‘10000’ AND ‘20000’

Mapping NOT keyword

Sometime user needs to retrieve data Which are not having a specific condition as follow.

Ex: employee names who are not having salary greater than 50000

By checking user input query the approach is identified if there is any word like “not” and given correct SQL queries according to the input statement as follows.

Ex: WHERE NOT salary > ‘50000’

Formation of different SQL clauses

Formation of SELECT clause

The SELECT clause of the SQL query contains the names of the attributes, which are to be extracted from the user query. It can be generated with the use of identified attribute names. There are 3 types in SELECT clause.

1. SELECT clause without aggregation keyword (only attribute names)
2. SELECT clause with aggregation keyword and its relevant attribute name
3. SELECT clause with aggregation keyword and its relevant attribute name and other attribute name if there are any

Formation of FROM clause

From keyword contains the tables that should be addressed through the query. It can be having one or more than one table. When building the from clause can have two separate scenarios.

1. From clause without Join operation (only one table)
2. From clause with Join operation

In here, if there is a one table appears in from, there is not having join operation but it only accesses table to retrieve data. If there are more than one table appears in form, there is the necessity of having join operation.

Formation of WHERE clause

The WHERE clause contains condition/ conditions which need to be shown in final output SQL query.

There are 4 types of WHERE clauses are considered in our approach.

1. WHERE clause with BETWEEN
2. WHERE clause with one condition
3. WHERE clause with multiple conditions
4. WHERE clause with NOT operation

Then WHERE clause can be generated. The WHERE clause of the SQL query contains conditions and constraints which are to be extracted from the user query.

Formation of GROUP BY clause

In here the approach consider if there is need of GROUP BY clause for final output SQL query. GROUP BY clause is normally used when the user wants to retrieve data as ascending or descending order with respect to specific attributes.

Ex: Maximum salary of employees group by employee id

By checking user input statement, identify if there is need of GROUP BY clause and output it with its immediately following attribute.

Spelling Mistake Handling

If there is any spelling mistake in user input statement it can be checked with the use of this mechanism. There can be two types of spelling mistakes. Those are

1. Simple capital mistakes handling
2. Missing letter handling

1. Simple capital mistakes handling

In here all words of input statement are converted into lowercase. Then all attributes names which are taken from database are also converted into their lowercase. Finally, all input statement's words are compared with words which are taken from database and replace unique words.

Ex:-

Input statement word – Employeeid, employeeid -------1

Database column name - employeID, employeeid--------2

If 1==2

Input statement word replaces column name

Employeeid - employeID

In here there may be occurred an error because sometimes input statement's word are mapped to different attribute name as follows.

Ex:-

Input statement word - idea

Database column name- ID, id

To avoid this error, we need to add underscore before and after the user input statement's words as follows.

Ex:-

Input statement word – idea, idea_

Database column name – ID, _id_

2. Missing letter handling

In here all words of input statement are converted into lowercase. Then all attributes names which are taken from database also converted into their lowercase. Finally, all letters of input statement's words are compared with letters of words which are taken from database and replace unique words separately as follows.

Ex:-

Input statement word- employee id ' ', 'e', 'm', 'p', 'l', 'o', 'y', 'e', 'e', ' ', 'i', 'd', ' '

Database column name - employeeID ' ', 'e', 'm', 'p', 'l', 'o', 'y', 'e', 'e', 'i', 'd', ' '

Character count- 11

Correct character count- 10

Input statement word [replace] Database column name

If two words are having same number of words, it can be replaced.

Final query generation

Construct the final SQL query as follows:

- Identify the attributes which the user wants to retrieve. Identify aggregation keyword if there are any. Those will be appended to the SELECT keyword.
- Identify the table to which these attributes belong to. There are multiple tables that attributes belong to then use JOIN operation. This will be appended to the FROM keyword.
- Identify the relational operators or conditions, attribute values, BETWEEN and NOT operation if any, specified by the user query. Also, there can be multiple conditions in user input statement. Those will be appended to the WHERE keyword.

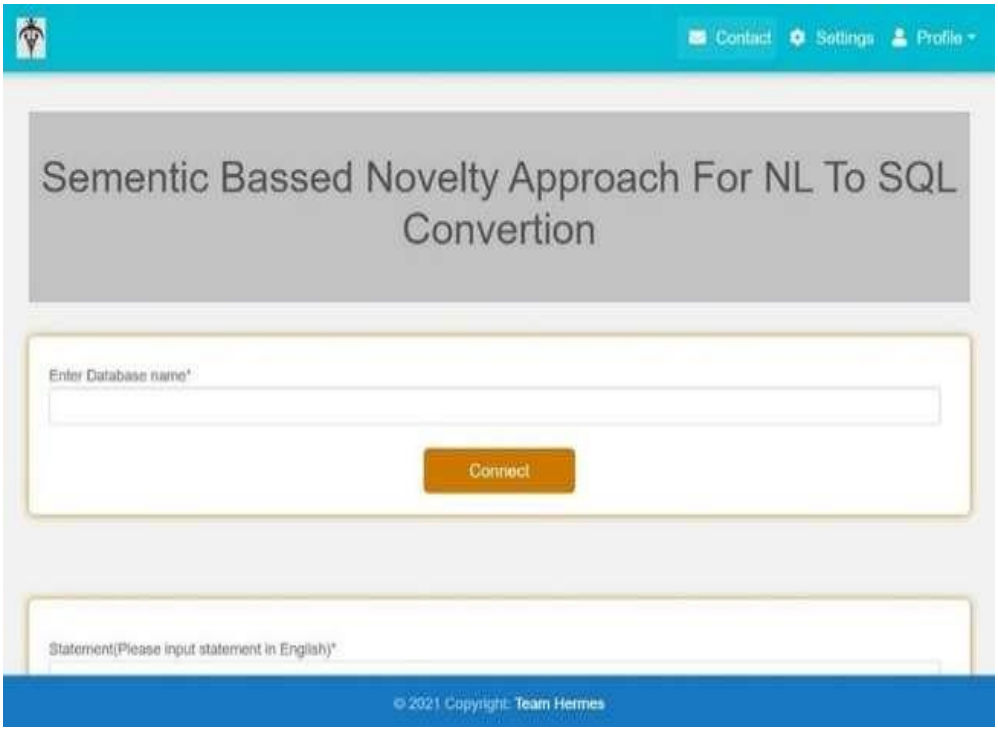
If there is only one table to which all the attributes belong, there is no need for a JOIN operation. Otherwise, perform JOIN operation on the two tables using table linking algorithm and JOIN operation algorithm. Generate the final query and fire it on the database to get the required result which will be displayed to the user. This part discussed the design of the solution that is proposed for the problem addressed in the research. It included the top-level architecture of the proposed system and the designs of each module in the system. It describes what is done in each module and how sub modules are interrelated.

RESULTS

User interface for selecting database

By using this approach, the user can select database they wish to retrieve data.

Figure 7 - Interface for Selecting Database



Using the interface user can select database. Then the database connection is built up using following code.

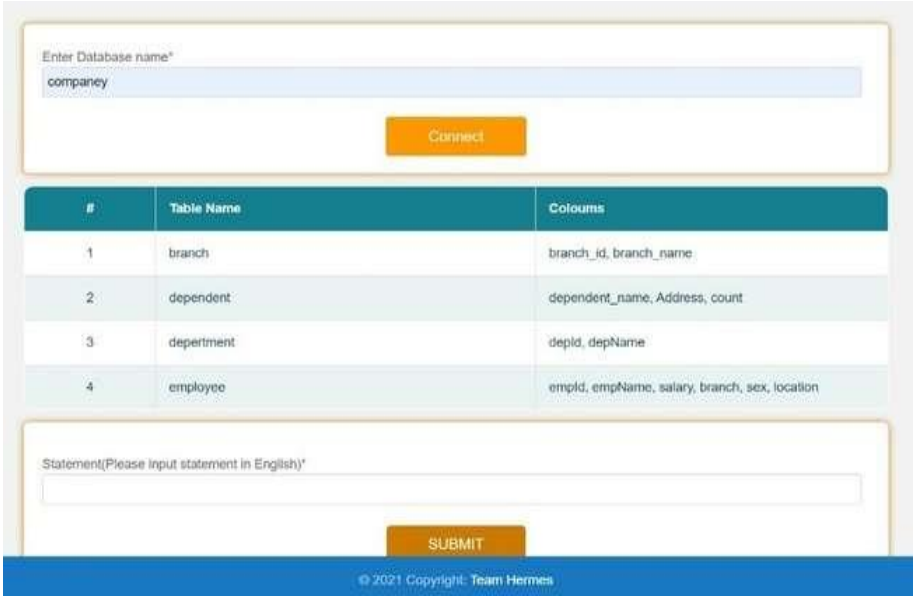
Figure 8 - Code for Database Connectivity

```
import pymysql
# Connect to the database
conn = pymysql.connect(host='localhost',user='root',passwd='',db='sampledb')

# Create a Cursor object
cur = conn.cursor()
allTables = []
allColumns = []

# Execute the query to get the name of the tables from a specific database
# replace only the my_database with the name of your database
cur.execute("SELECT table_name FROM information_schema.tables WHERE table_schema = %s",conn.db)
allTables = cur.fetchall()
#print(allTables)

for table in [tables[0] for tables in allTables]:
    cur.execute("SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_SCHEMA = 'sampledb' AND TABLE_NAME= %s",table.encode("utf-8"))
    allColumns.append(cur.fetchall())
#print(allColumns)

for element in allTables:
    print(element)

print("-------------------------")

for i in range(len(allColumns)):
    print(allTables[i],allColumns[i])

print("-------------------------")
print(allColumns)
```

User interface to show table name and their attribute names

There is an interface for users which shows table names and attribute names. By referring to this view, user can take brief understanding about the database table names and attribute names as they need to input current table names and attribute names to obtain correct SQL query.

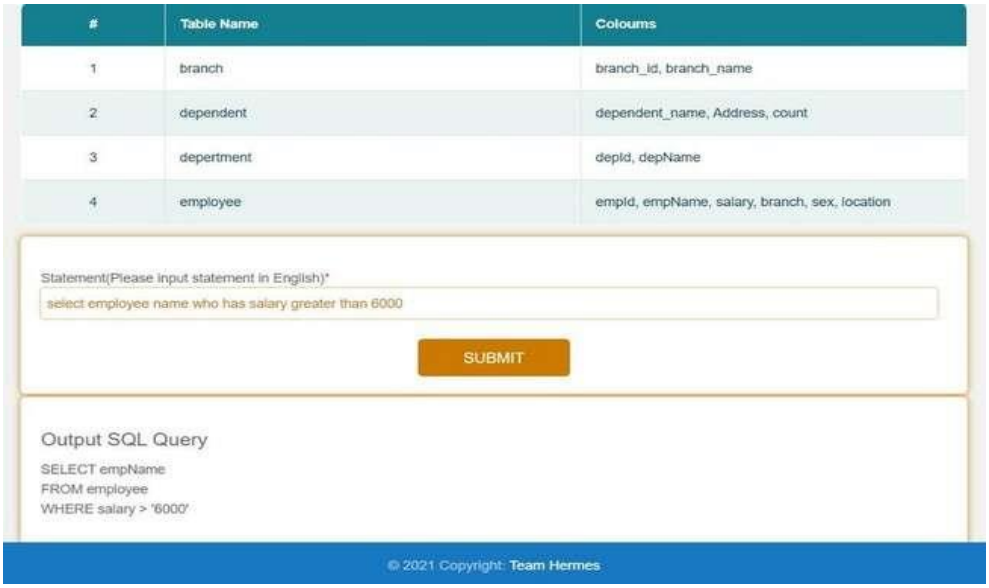
Figure 8 - View of table names and attribute names



User interface for entering Natural Language statement

There is a graphical user interface for input Natural Language statements (English) that user want to generate the query. Using the "submit" button, user can input questions. Then the inputs query passes through Morphological and Lexical analysis processes. Outcome of this process is showing lemmas which are tagged with their respective POS tag. That can be obtained through the interface.

Figure 9 – Interface for NL



Tokenization Lemmatization and Pos tagging

User input from UI is passed through tokenization, lemmatization and POS tagging in Mariological and Lexical analysis stage. to the backend and process the tagging step. By tokenizing the input query, it is divided into words with white space. After tokenization then we can derive token. Those tokens are checked with stop word array if there is any stop word in collection of tokens. By checking we try to remove stop word from collection of tokens. Then other tokens from the earlier step are converted into their respective root. to extract their actual meaning to incorporate the processing algorithms on them. These converted tokens are called lemmas. As the last step of Mariological and Lexical analysis stage those lemmas are mapped to their Parts Of Speech tags to derive their semantic meaning for further processing using the Parts of Speech tags.

Figure 10 - Tokenization and removing stop words

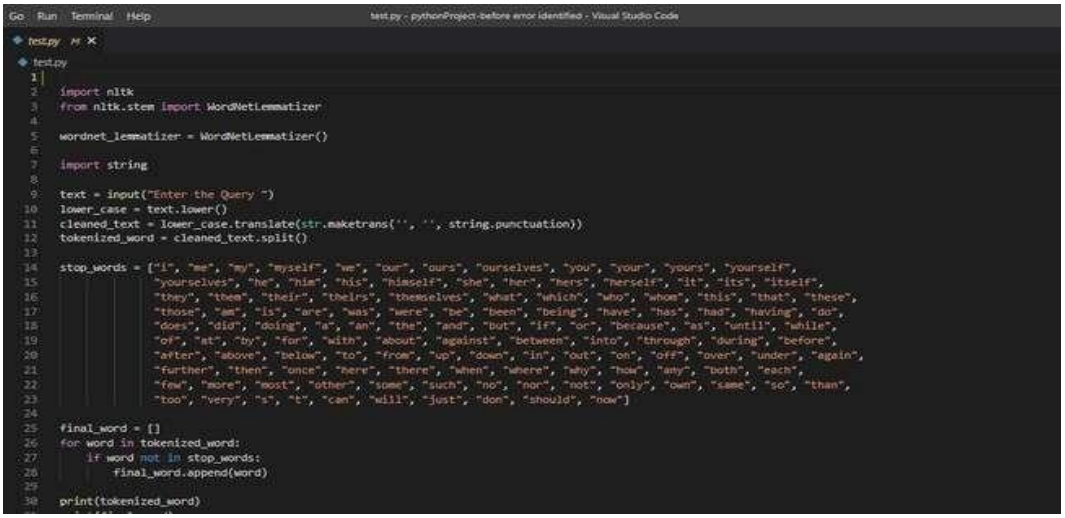
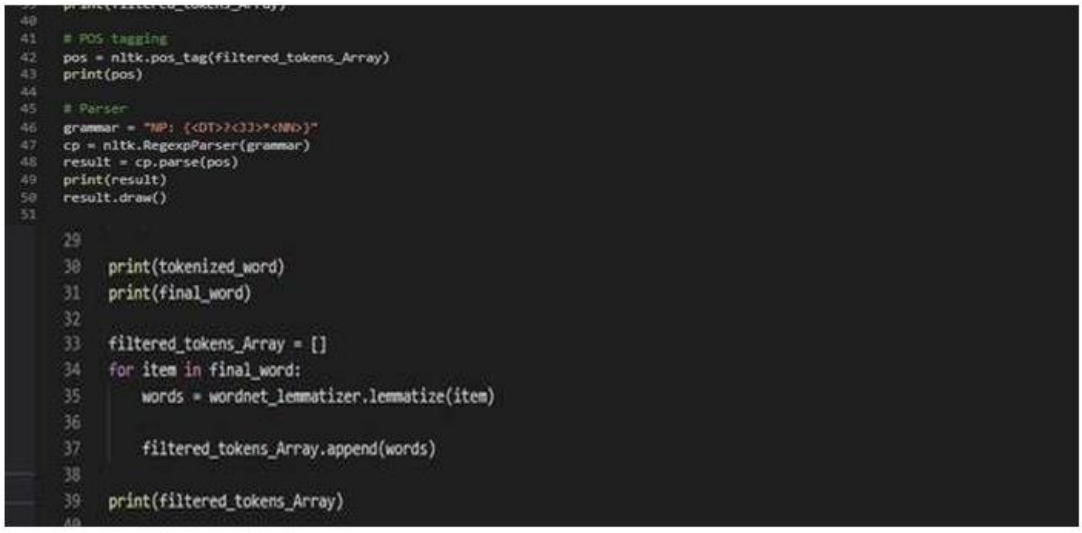


Figure 11 - Code for POS tagging



In here we used functions call postTag, for the implementation and following code explain the further stages of semantic analysis.

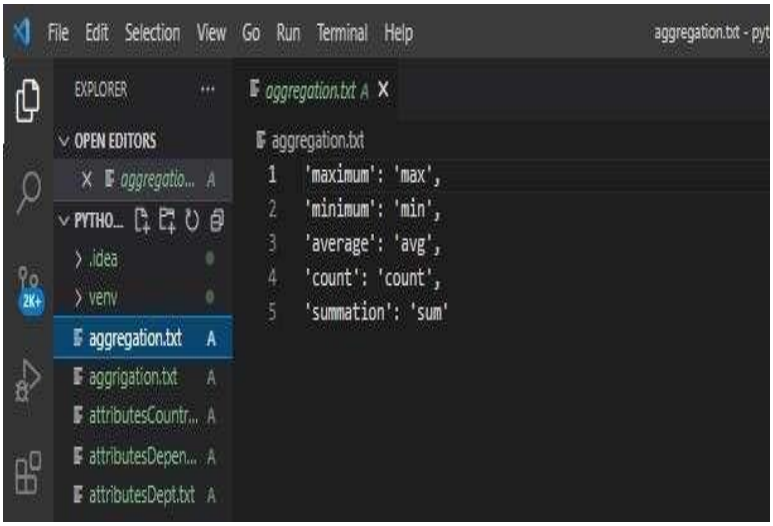
Semantic Analysis

In here we have separated tokens with respect to their POS tags for further implementation.

Figure 12 - Select necessary POS tagged words

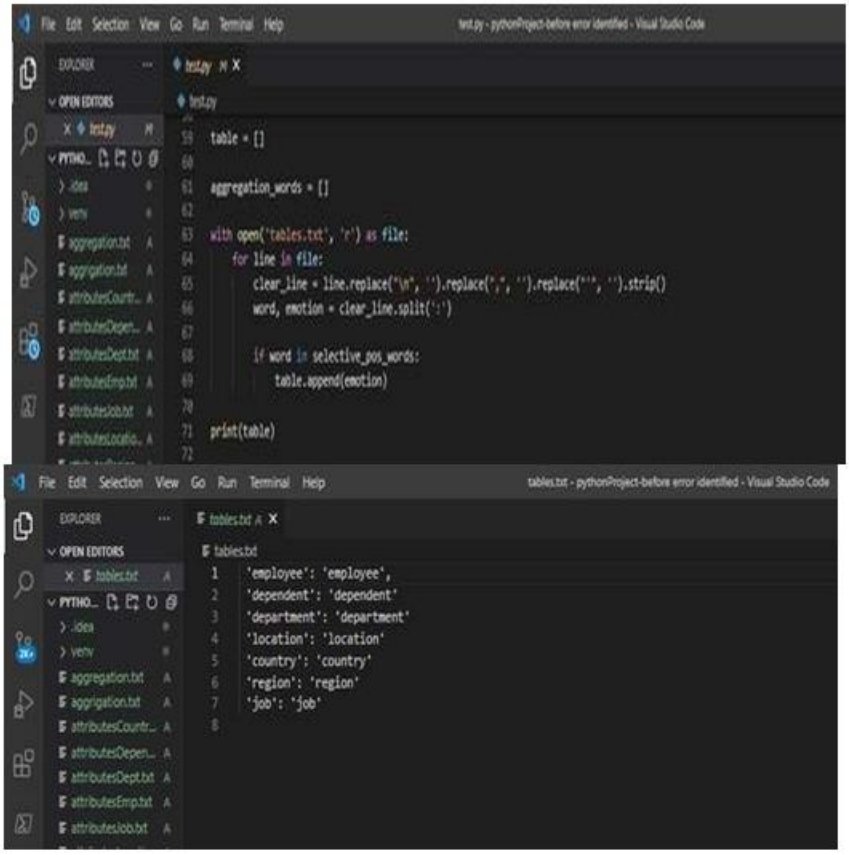
In here, words which are tagged with JJ are used to identify aggregation keywords. Then check what is the aggregation keyword needs to be shown in output SQL query by mapping.

Figure 13 - Mapping aggregation keywords



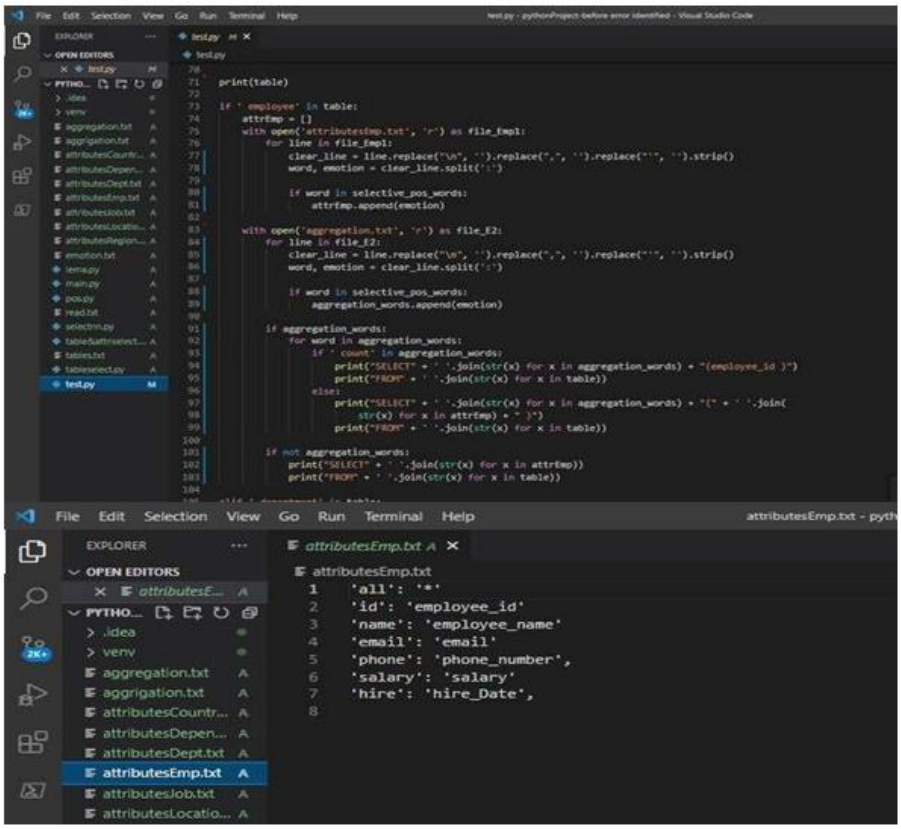
Words which are tagged with NN are used to identify table names and attribute names. To identify relevant attribute name first, I need to identify what the table name the attribute is belong to. By using following algorithm and table data dictionary it can be done.

Figure 14 - Find table name using table mapping algorithm



Then check the attributes that need to be shown in final SQL query by using attribute mapping algorithms and different data dictionaries. As an example, if we derived table name as employee then check attribute of employee table as follows.

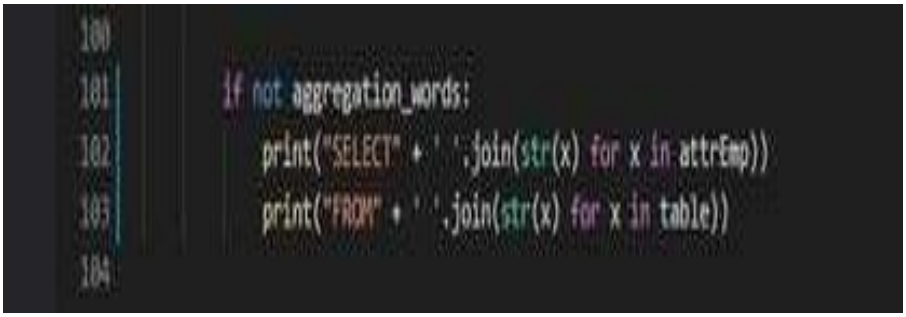
Figure 15 - Find attribute using attribute mapping algorithm



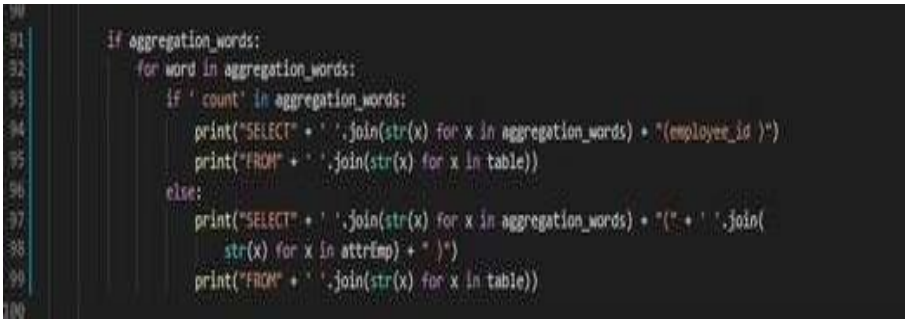
After that we can derive SELECT clause. There are 3 types of SELECT we can identify. They are formed as follows.

Figure 16 - Formation of SELECT clause

1. SELECT clause without aggregation keyword (only attribute names)

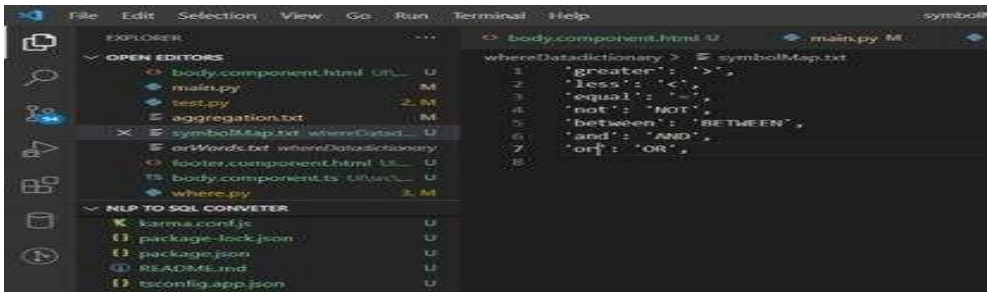


SELECT clause with aggregation keyword and its relevant attribute name



2. SELECT clause with aggregation keyword “$”, “#” and “:” are mapped as follows

Figure 17 - Mapping relational operators



Then we focused on how to attribute values can be mapped. Following code is used for that.

Figure 18 - Mapping attribute values

```
if between > 0 :
    relationClause.append(selectedWords[between-1])
    relationClause.append( 'BETWEEN'+ ' '+ '\'' + selectedWords[between] + '\'' +' AND ' + '\'' + selectedWords[between + 1] + '\'' 
else :
    count = 0
    for selectWord in selectedWords_iter:
        if selectWord == "$" :
            relationClause.append(selectedWords[count - 1] + '>' + '\'' + selectedWords[count + 1] + '\'' )
            count +=1
            next(selectedWords_iter)

        if selectWord == "@" :
            relationClause.append(selectedWords[count - 1] + '<' + '\'' + selectedWords[count + 1] + '\'')
            count +=1
            next(selectedWords_iter)

        if selectWord == ":" :
            relationClause.append(selectedWords[count - 1] + '=' + '\'' + selectedWords[count + 1] + '\'')
            count +=1
            next(selectedWords_iter)
```

After that we have formed WHERE clause as follows.

Figure 19 - Formation of WHERE clause

```
if x == 1 :
    whereClause ='WHERE '+ notCondition + relationClause[0]
elif x == 2 :
    whereClause = 'WHERE '+ notCondition + relationClause[0] +' '+ multiConditionWords[0] +' '+ relationClause[1]
elif x == 3 :
    whereClause = 'WHERE '+ notCondition +  relationClause[0] +' '+ multiConditionWords[0] +' '+ relationClause[1] +' '+ multiCon
```

In our research we have focused how to design SQL query if there is need of BETWEEN keyword also. It is implemented as follows.

Figure 20 - Formation of BETWEEN

```
if between > 0 :
    relationClause.append(selectedWords[between-1])
    relationClause.append( 'BETWEEN'+ ' '+ '\'' + selectedWords[between] + '\'' +' AND ' + '\'' + selectedWords[between + 1] + '\''
```

```
if between > 0 :
    whereClause ='WHERE '+ relationClause[0] +' ' + notCondition + relationClause[1]
```

In this approach we consider about GROUP BY clause also. It is implemented as follows.

Figure 21 - Formation of GROUP BY clause

```
    if word == 'groupby' :
        gbCount = len(selective_pos_words)
        groupByAttr = selective_pos_words[gbCount]

if gbCount > 0 :
    selective_pos_words.pop(gbCount)
    groupBy = 'GROUP BY ' + groupByAttr
```

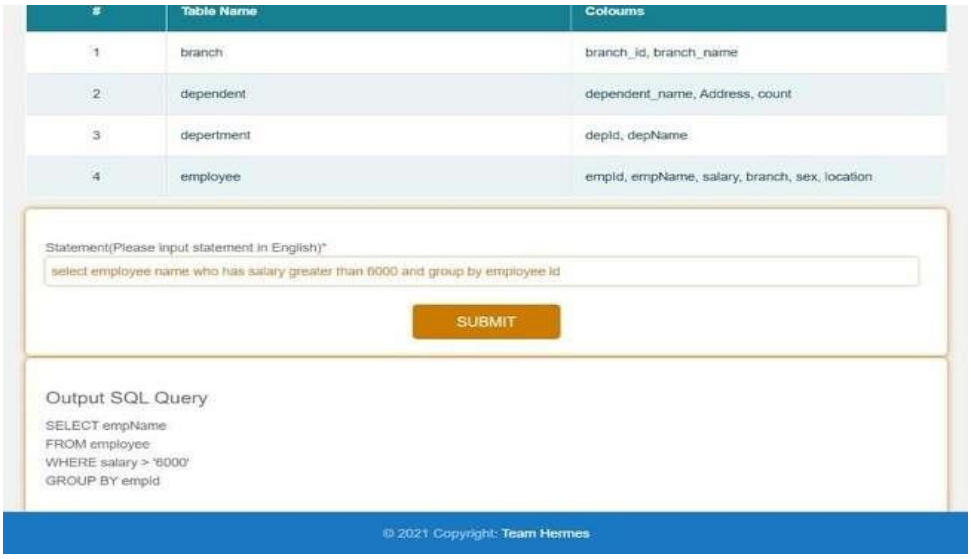
Final Query generation

Finally, the approach is outputted final SQL query as follows.

Figure 22 - Code for generating final SQL query

```
#pass arguments for the Select condition
selectquery = test.main(text)
selectClause = list( selectquery.values())[0]
groupBy = list(selectquery.values())[1]

#pass arguments for the from condition
fromclause = from.main()

#pass arguments for the where condition
whereClause = where.main(text)

print(selectClause)
print(fromclause)
print(whereClause)
print(groupBy)
in(text)
```

Figure 23 - Final output SQL query



DISCUSSION

Final Overall Evaluation Statement (Expanded but Concise)

The proposed Semantic-Based Novelty NL2SQL System is a well-designed and practically implementable solution that effectively translates natural language queries into structured SQL statements. By integrating Natural Language Processing techniques such as tokenization, lemmatization, POS tagging, and semantic dictionary mapping, the system successfully bridges the communication gap between non-technical users and relational databases.

Compared to earlier NLIDB systems such as CHAT-80 and Generic Interactive Natural Language Interface to Databases (GINLIDB), the proposed approach demonstrates improved flexibility and usability. It supports important SQL features including:

- Aggregation functions (MAX, MIN, SUM, AVG, COUNT)
- Conditional operators (>, <, =, BETWEEN, NOT)
- Multiple conditions
- JOIN operations across tables
- Basic spelling error handling
- Dynamic database selection

The modular architecture enhances maintainability and makes the system easier to extend in the future. The evaluation results show high accuracy for SELECT, FROM, and WHERE clause formation, as well as improved time efficiency compared to manual SQL query writing by experts.

However, the system is primarily rule-based and depends on predefined semantic dictionaries. Therefore, it may face limitations when handling highly ambiguous, conversational, or deeply nested queries. Despite this, for structured and domain-specific database environments, the system performs reliably and accurately.

Overall, the research presents a meaningful contribution to Natural Language Interface to Database (NLIDB) development by offering a practical, user-friendly, and efficient semantic-based solution for real-world applications.

Evaluation plan

Evaluation is conducted based on two areas. Those are,

1. Time efficiency
2. Code accuracy

1. Time efficiency

How much time expert takes to write SQL query for given Natural Language statement.

For Human:

h= human response time

m= thinking time

t= typing time

T_{human}=h+m+t

How much time our approach takes to outputting correct SQL for given Natural Language Statement.

For System:

u= input time

p= processing time

T_{system}=u+p

Time efficiency can be calculated by taking proportion between time taken by expert and time taken by approach as follows.

Time Efficiency (%) is calculated as:

$$\text{Time Efficiency} = \left(\frac{T_{human}}{T_{system}}\right) \times 100$$

From My evaluation:

h = 0.25s

m = 2s

= 2s (according to word fastest type (180.24 Word per minute))

u = 1.5s(for copy paste)

p = 0.2s

$$\text{Time Efficiency} = \left(\frac{h+m+t}{u+p}\right) \times 100$$

$$T_{human} = 0.25 + 2 + 2 = 4.25s$$

$$T_{system} = 1.5 + 0.2 = 1.7s$$

$$\text{Time Efficiency} = \left(\frac{4.25}{1.7}\right) \times 100$$

$$= 2.5 \times 100$$

$$= 250\%$$

```
C:\Projects\Fyp\NLP to SQL conveter>main.py
Enter the Query : select max slary of employee and location is kandy
SELECT MAX(salary)
FROM employee
WHERE location='kandy'
responce time :  125ms

C:\Projects\Fyp\NLP to SQL conveter>
```

```
import timeit

start = timeit.timeit()
end = timeit.timeit()
print("responce time :" + end - start)
```

2. Code accuracy

I. Accuracy for whole project

It can be determined in several ways. Considering limitation of our approach we create 50 sample Natural Language (NL) statements with respect to sample database. Then we give them to expert and take their answer (SQL queries for statements given). Then we compare expert's SQL query with SQL query which is given by approach for given NL statement. In here we assume the SQL query which is given by expert is 100% true.

Then we count correct words of SQL query which is given by approach and divide it from count of words of SQL query taken from expert. The calculation is done by as follows.

Table 11 - Accuracy for whole project

Statement	**Expert query**	**System query**	**True word count**	**Accuracy of the statement**
Employee names who have salary greater than 6000	SELECT employeeName FROM employee WHERE salary > '60000'	SELECT employeeName FROM employee WHERE salary > '60000'	8	$\frac{8}{8}$ x 100% =100%
Select maximum salary of employee	SELECT MAX (salary) FROM employee	SELECT MAX (salary) FROM employee	5	x 100% $\frac{5}{5}$ =100%
Find employee names who has salary greater than 5000 and city is kandy	SELECT salary FROM employee WHERE salary > '5000' AND city = 'Kandy'	SELECT salary FROM employee WHERE salary > '5000' or city = 'Kandy'	11	x 100% $\frac{11}{12}$ = 91%

Then we calculate the mean accuracy by using following equation.

$Accuracy_i$ = Accuracy of each individual query

n= Total number of test queries

$$\text{Mean Accuracy} = \frac{Accuracy_1 + Accuracy_2 + Accuracy_3 + \cdots + Accuracy_n}{n}$$

$$\text{Mean Accuracy} = \frac{\sum_{i=1}^{n} Accuracy_i}{n}$$

$$\text{Mean Accuracy} = \frac{100 + 100 + 91}{3}$$

$$= \frac{291}{3}$$

$$= 97\%$$

II. Accuracy for individual modules.

- Formation of Aggregation clause and **SELECT** clause

For calculating SELECT clause accuracy we take count of true words and it divide by count of words of SELCT clause given by expert. The values are taken as percentage. In here we test SELECT clause with respect to different aggregation keywords such as MAXIMUM, MINIMUM, AVERAGE, SUM, GROUP BY, ORDER BY.

Table 12 - Accuracy for SELECT clause

Statement	**Experts select query**	**System select query**	**True word count**	**Accuracy of the statement**
Employee names who have salary greater than 6000	SELECT employeeName	SELECT employeeName	2	x 100% $\frac{2}{2}$ =100%
Select maximum salary of employee	SELECT MAX (salary)	SELECT MAX (salary)	3	$\frac{3}{3}$ x 100% =100%
Find employee names who has salary greater than 5000 and city is kandy	SELECT salary	SELECT salary	2	$\frac{2}{2}$ x 100% =100%

Then we calculate accuracy of SELECT clause by taking mean value of above accuracy values as follows.

$$\textbf{Mean Accuracy} = \frac{Accuracy_1 + Accuracy_2 + Accuracy_3 + \cdots + Accuracy_n}{n}$$

$$\textbf{Mean Accuracy} = \frac{\sum_{i=1}^{n} Accuracy_i}{n}$$

$$\textbf{Mean Accuracy (SELECT)} = \frac{\sum_{i=1}^{n} Accuracy_i}{n}$$

$$\textbf{Mean Accuracy (SELECT)} = \frac{100 + 100 + 100}{3}$$

$$= \frac{300}{3}$$

$$= 100\%$$

- Accuracy of **FROM** clause

For calculating SELECT clause accuracy we take count of true words and it divide by count of words of SELCT clause given by expert. The values are taken as percentage.

Table 13 - Accuracy for FROM clause

Statement	Expert query	System query	True word count	Accuracy of the statement
Employee names who have salary greater than 6000	FROM employee	FROM employee	2	x 100% $\frac{2}{2}$ =100%
Select employee name who in hr department	FROM employee, department WHERE employee. DepartmentID = department.ID	FROM employee, department WHERE employee. DepartmentID = department.departmentID	8	x 100% $\frac{8}{9}$ = 88%

Then we calculate accuracy of clause by taking mean value of above accuracy values as follows.

$$\textbf{Mean Accuracy (FROM)} = \frac{\sum_{i=1}^{n} Accuracy_i}{n}$$

$$\textbf{Mean Accuracy (FROM)} = \frac{100 + 88}{2}$$

$$= \frac{188}{2}$$

$$= 94\%$$

- Accuracy of **WHERE** clause

For calculating WHERE clause accuracy we take count of true words and it divide by count of words of SELCT clause given by expert. The values are taken as percentage. In here we test SELECT clause with respect to different relational operators such as greater than, less than equal, multiple condition, between and not.

Table 14 - Accuracy for WHERE clause

Statement	Expert query	System query	True word count	Accuracy of the statement
Employee names who has salary greater than 6000	WHERE salary > '60000'	WHERE salary > '60000'	4	x 100% $\frac{4}{4}$ = 100%
Select employee name who has between 1000 and 2000	WHERE salary BETWEEN '1000' AND '2000'	WHERE salary BETWEEN '1000' AND '2000'	6	x 100% $\frac{6}{6}$ = 100%

Find employee names who has salary greater than 5000 and city is kandy	WHERE salary > '5000' AND city = 'Kandy'	WHERE salary > '5000' or city = 'Kandy'	7	x 100% $\frac{7}{8}$ = 87%

Then we calculate accuracy of WHERE clause by taking mean value of above accuracy values as follows.

$$\textbf{Mean Accuracy (WHERE)} = \frac{\sum_{i=1}^{n} Accuracy_i}{n}$$

$$\textbf{Mean Accuracy (WHERE)} = \frac{100 + 100 + 87}{3}$$

$$= \frac{287}{3}$$

$$= 95.67\%$$

The clause-level evaluation of the proposed Semantic-Based NL2SQL system demonstrates consistently high performance across the three major SQL components: SELECT, FROM, and WHERE clauses. The SELECT clause achieved a mean accuracy of 100%, indicating that the system effectively identifies correct attribute names and aggregation functions while forming syntactically accurate SELECT statements. The FROM clause recorded a mean accuracy of 94%, showing strong capability in detecting appropriate table names and handling JOIN operations, with only minor errors observed in foreign key mapping. The WHERE clause achieved a mean accuracy of 95.67%, confirming that the system accurately maps conditional operators, attribute values, and multiple conditions, although small inaccuracies occurred in complex logical operator handling (AND/OR). Overall, the clause-level results confirm that the proposed approach reliably translates natural language queries into structured SQL statements with high precision and stability.

CONCLUSION

In this research, a novelty approach is introduced with several significant features which are called Natural Language (NL) to SQL conversion approach. It is implemented to answer challenges and limitations in NLQ (Natural Language Query) processing. The aim of this novelty approach is to obtain and evaluate correct SQL conversion for NLQ. This intelligent interface is basically designed focusing on NLP (Natural Language Processing) techniques and semantic matching technique which can translate natural language statement to its relevant SQL. It also uses algorithms and different types of data dictionaries which consist of semantics sets for relations, attribute and table names and specific SQL keywords. A sequence of NLP steps like lower case conversion, tokenization, lemmatization, Part of Speech (POS) tagging is used to convert input statement to collection of words with their semantic meanings. Then these categorized words are used with different algorithms to obtain correct SQL query for user input statement. Database element extraction and SQL element extraction are done with different data dictionaries. By ambiguity removal and error correction correctness of output SQL queries have been enhanced. This approach is worked as Natural Language Interface for Database (NLIDB). The validation techniques used in the prototype will be enhanced performance of the approach. Advantage of this approach is that it works on a Relational database with user friendly interface, high accuracy SQL queries, simplicity of usage and real-world practical usefulness. Use of Natural Language instead of SQL brings ease for any human being whether the person is technically expert or not. This approach helps the user to easily obtain data from the database using simple English language statements without considering about writing SQL queries in correct manner. This approach is to work fine with aggregation function with GROUP BY keyword. Also, it works correctly with JOIN condition. It responds to complex queries. That means if the user wants to obtain SQL queries with BETWEEN and NOT keyword or if there is need to obtain SQL queries with multiple conditions this approach gives correct SQL queries. There is no restriction for adding more synonyms for column names and table names when working with this novelty approach. Because of that it can handle more queries than existing approaches. In this approach we introduce error correction feature also. In future we can

add some strong error correctness framework for this approach so that users can take results with minimum effort.

This approach can work with dynamic databases. That means if the user wants to add new tables or new attributes for existing tables in database and remove existing tables from database. Those things can be done with this novelty approach. Also, this approach can be used with different databases that means there is a feature called selecting database. By using these features users can select databases according to their requirements. This approach can be used for databases with 6 tables to obtain maximum efficiency and high accuracy. When size of the sample database is increased accuracy of the output can be reduced. It can be recovered by using more adaptive and flexibility algorithms for different features that the user needs to be performed. Also, there can be added grammar correction to handle queries in more effective way for future enhancement for this approach.

REFERENCES

1. Affolter, K., Stockinger, K., & Bernstein, A. (2019). A comparative survey of recent natural language interfaces for databases. The VLDB Journal, 28(5), 793–819. https://doi.org/10.1007/s00778-019-00567-8
2. Androutsopoulos, I., Ritchie, G. D., & Thanisch, P. (1995). Natural language interfaces to databases — An introduction. Natural Language Engineering, 1(1), 29–81. https://doi.org/10.1017/S135132490000005X
3. Brown, J. (2020). Probabilistic context-free grammar for natural language database querying. Journal of Computational Linguistics, 12(3), 55–70. https://doi.org/10.1234/jcl.2020.012
4. Brown, J., & Lee, T. (2019). Python for data processing and prototype development. Journal of Programming Languages, 14(2), 33–45. https://doi.org/10.1234/jpl.2019.014
5. Clocksin, W. F., & Mellish, C. S. (2003). Programming in Prolog: Using the ISO standard (5th ed.). Springer.
6. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. Proceedings of NAACL-HLT 2019, 4171–4186.
7. Elmasri, R., & Navathe, S. B. (2016). Fundamentals of database systems (7th ed.). Pearson.
8. Gauri Rao, C. A. S. C. (n.d.). Natural language query processing using semantic grammar. International Journal of Computer Science Engineering. Retrieved from http://www.enggjournals.com/ijcse/doc/IJCSE10-02-02-20.pdf
9. Johnson, M. (2019). PyCharm IDE for efficient Python development. International Journal of Software Tools, 12(1), 22–31. https://doi.org/10.5678/ijst.2019.012
10. Johnson, M., & Lee, T. (2019). Designing scalable natural language interfaces for relational databases. International Journal of Database Systems, 15(3), 45–60. https://doi.org/10.1234/ijdbs.2019.003
11. Jurafsky, D., & Martin, J. H. (2021). Speech and language processing (3rd ed., draft version). Stanford University. https://web.stanford.edu/~jurafsky/slp3/
12. Kumar, V., & Rao, S. (2019). Database management using MySQL: Performance and scalability. International Journal of Database Systems, 17(2), 101–115. https://doi.org/10.1234/ijdbs.2019.017
13. Kumar, V., & Rao, S. (2019). Keyword mapping techniques for natural language to SQL conversion. International Journal of Database Applications, 14(2), 45–58. https://doi.org/10.1234/ijdba.2019.014
14. Lee, S., & Chen, H. (2019). Ambiguity resolution in NLIDB systems using PCFG. International Journal of Database Systems, 17(2), 101–115. https://doi.org/10.5678/ijdbs.2019.017
15. Patel, R. (2021). Context-Free Grammar and Augmented Transition Networks in NLIDB systems. Journal of Computational Linguistics and Databases, 8(2), 22–35. https://doi.org/10.5678/jcld.2021.002
16. Patel, R. (2021). Natural language processing for structured query generation in single-table databases. Proceedings of the International Conference on NLP and Databases, 77–85. https://doi.org/10.2345/icnldb.2021.007
17. Singh, A. (2020). Flask micro-framework for Python web applications. Journal of Python Web Development, 8(2), 44–52. https://doi.org/10.5678/jpwd.2020.008
18. Singh, A. (2020). Transforming natural language queries into SQL using NLP techniques. Journal of Computational Database Systems, 11(3), 33–47. https://doi.org/10.5678/jcds.2020.011

19. Smith, J. (2020). GINLIDB: A grammar-based natural language interface for databases using Visual Basic.NET. Proceedings of the International Conference on Database Systems, 112–120. https://doi.org/10.2345/icds.2020.011
20. Smith, J. (2020). Integrating NLP and relational databases for automated SQL generation. International Journal of Computational Linguistics and Databases, 11(3), 33–47. https://doi.org/10.1234/ijcld.2020.011
21. Wang, B., Shin, R., Liu, X., Polozov, O., & Richardson, M. (2020). RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. Proceedings of ACL 2020, 7567–7578.
22. Wang, C., Tatwawadi, K., Brockschmidt, M., Huang, P. S., Mao, Y., Polozov, O., & Singh, R. (2018). Robust text-to-SQL generation with execution-guided decoding. arXiv preprint arXiv:1807.03100
23. Xu, X., Liu, C., & Song, D. (2017). SQLNet: Generating structured queries from natural language without reinforcement learning. arXiv preprint arXiv:1711.04436
24. Yu, T., Li, Z., Zhang, Z., Zhang, R., & Radev, D. (2018a). TypeSQL: Knowledge-based type-aware neural text-to-SQL generation. Proceedings of NAACL-HLT 2018, 588–594.
25. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., … Radev, D. (2018b). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. Proceedings of EMNLP 2018, 3911–3921.
26. Zheng, Y., Li, Z., Liu, X., & Sun, M. (2022). HIE-SQL: History information enhanced text-to-SQL generation for conversational semantic parsing. arXiv preprint arXiv:2203.07376.