

A
Project Report
on

NATURAL LANGUAGE TO SQL GENERATOR

Submitted in partial fulfillment of the requirements for the award of the degree of
Bachelor of Technology

by

Poddutoori Neetha Reddy
(20EG105435)

Madasu Surya Teja
(20EG105423)



Under the guidance of

Ravinder Reddy B

Assistant Professor,

Department of CSE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ANURAG UNIVERSITY

VENTAKAPUR (V), GHATKESAR (M), MEDCHAL (D), T.S - 500088

TELANGANA

(2023-2024)



CERTIFICATE

This is to certify that the project report entitled “**Natural Language To SQL Generator**” being submitted by **Poddutoori Neetha Reddy** bearing the hall ticket number **20EG105435**, **Madasu Surya Teja** bearing the hall ticket number **20EG105423** respectively in partial fulfillment of the requirements for the award of the degree of the **Bachelor of Technology in Computer Science and Engineering** to **Anurag University** is a record of bonafide work carried out by them under my guidance and supervision from 2023 to 2024.

The results presented in this report have been verified and found to be satisfactory. The results embodied in this report have not been submitted to any other University for the award of any other degree or diploma.

Signature of The Supervisor
B. Ravinder Reddy
Assistant Professor, Dept of CSE
Anurag University

Dr. G. Vishnu Murthy
Professor
Dean, Department of CSE
Anurag University

External Examiner

DECLARATION

I hereby declare that the report entitled “**Natural Language To SQL Generator**” submitted to the **Anurag University** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology (B. Tech)** in **Computer Science and Engineering** is a record of an original work done by us under the guidance of **Ravinder Reddy B, Assistant Professor** and this report has not been submitted to any other university for the award of any other degree or diploma.

Date:

Place: Anurag University, Hyderabad

Poddutoori Neetha Reddy

(20EG105435)

Madasu Surya Teja

(20EG105423)

ACKNOWLEDGMENT

We would like to express our sincere thanks and deep sense of gratitude to project supervisor **RAVINDER REDDY B**, Assistant Professor, Department of CSE for his constant encouragement and inspiring guidance without which this project could not have been completed. His critical reviews and constructive comments improved our grasp of the subject and steered to the fruitful completion of the work. His patience, guidance and encouragement made this project possible.

We would like to acknowledge our sincere gratitude for the support extended by **Dr. G. VISHNU MURTHY**, Dean, Department of CSE, Anurag University. We also express my deep sense of gratitude to **Dr. V. V. S. S. S. BALARAM**, Academic Coordinator. **Dr. PALLAM RAVI**, Project Coordinator and Project review committee members, whose research expertise and commitment to the highest standards continuously motivated me during the crucial stage of our project work.

We would like to express our special thanks to **Dr. V. VIJAYA KUMAR**, Dean School of Engineering, Anurag University, for his encouragement and timely support in our B.Tech program.

Poddutoori Neetha Reddy
(20eg105435)

Madasu Surya Teja
(20eg105423)

ABSTRACT

In the contemporary landscape, accessing databases poses challenges for untrained individuals due to technical complexities. Natural Language to SQL (Structured Query Language) generator models have alleviated these obstacles but struggle with extensive databases, often leading to inaccuracies stemming from inadequate training data. A proposed solution introduces a novel model merging OpenAI models with Langchain technology. This innovative approach aims to overcome limitations by harnessing OpenAI's fine-tuned GPT-3.5-turbo model capabilities alongside Langchain's rapid processing, enabling efficient handling of vast database information. By leveraging OpenAI's advanced large language models and Langchain's high-speed processing, this model seeks to bridge the gap for non-technical users, swiftly converting natural language queries into precise SQL commands. Its strength lies in accurately interpreting queries even with massive datasets, promising enhanced database accessibility for individuals lacking technical expertise. This innovation could foster inclusivity and user-friendly data utilization by making databases more accessible.

Keywords:- Natural language, SQL, Langchain technology, OpenAI, Queries

TABLE OF CONTENT

S. No.	CONTENT	Page No.
	Abstract	I
	List of Figures	II
	List of Screenshots	III
	List of Tables	IV
	List of Abbreviations	
1.	Introduction	1
	1.1. Structured Query Language	1
	1.2. Natural Language Processing	3
	1.3. Motivation	4
	1.4. Problem Definition	4
	1.5. Problem Illustration	5
	1.6. Objective of the Project	6
2.	Literature Document	7
3.	Natural Language To SQL Generator	15
	3.1. Fine-tuning the model	15
	3.2. Generating summaries using fine-tuned model	18
	3.3. Illustration- SQL Generation	24
4.	Design	26
	4.1. Introduction	26
	4.2. Use case diagram	26
	4.3. Class diagram	27
	4.4. Activity diagram	28
	4.5. Sequence diagram	29
5.	Implementation	31
	5.1. Functionalities	31

5.2. Attributes	33
5.3. Experimental Screenshot	35
5.4. Dataset	35
6. Experimental Setup	36
6.1. Obtain OpenAI API key	36
6.2. Setup Python IDLE	36
6.3. Setup Streamlit	37
6.4. Libraries used	39
6.5. Parameters	43
7. Discussion of Results	45
8. Conclusion	49
9. Future Enhancements	50
10. References	51

List of Figures

Figure No.	Figure Name	Page No.
1.1	Accuracy v/s Input Size	4
3.1	Overview of fine-tuning procedure	10
3.2	SQL Generation Example	20
4.1	Use case diagram	21
4.2	Class diagram	22
4.3	Activity diagram	23
4.4	Sequence diagram	24
7.1	Precision score of models	40
7.2	Execution Accuracy of models	41
7.3	Semantic Accuracy of the models	42

List of Screenshots

Screenshot No.	Screenshot Name	Page No.
3.1	JSON lines dataset	11
3.2	Details of fine-tuned model on OpenAI account	12
5.1	Database URI and Question	30
5.2	Output	30
5.3	Dataset	36
6.1	Coding environment screenshot	36
6.2	User Interface	37
6.3	Database	37

List of Tables

Table No.	Table Name	Page No.
1.1	Table Limitation problem in Existing Approach	3
2.1	Comparison of Existing Methods	9
7.1	Average precision score of different model	39
7.2	Average execution accuracy of different model	40
7.3	Average semantic accuracy of different model	41

List of Abbreviations

Abbreviations	Full Form
SQL	Structured Query Language
GPT	Generative Pre-trained Transformer
RDBMS	Relational Database Management Systems
GNN	Graph Neural Network
NLP	Natural Language Processing
Seq2Seq	Sequence-to-Sequence Model
NLIDB	Natural Language Interfaces to Databases
MAML	Model-Agnostic Meta-Learning
KB	Knowledge Base
ILP	Inductive Logical Programming
M-SQL	Multi-task Structured Query Language
BERT	Bidirectional Encoder Representations Transformer
LQL	Logical Query Language

EFFCN	Efficiently Compliant Flexible Compliant Natural language
LLM	Large Language Model
API	Application Program Interface
CSV	Comma-Separated Values
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
UML	Unified Modeling Language
LLSG	Large Language model based SQL Generation (Proposed Method)

1. Introduction

In the era of big data, where businesses rely heavily on data-driven decision-making, accessing and analyzing data efficiently is paramount. However, for many individuals, querying databases using Structured Query Language (SQL) can be a daunting task, often requiring specialized knowledge and training. Without SQL knowledge, individuals may have difficulty understanding the structure of the database, including tables, columns, relationships, and constraints. This makes it challenging to navigate and interpret the data effectively. SQL requires capabilities for data manipulation, such as filtering, sorting, aggregating, and joining datasets and without SQL knowledge, individuals may struggle to perform these tasks efficiently, leading to limitations in data analysis and reporting. So, the idea of creating a special system to generate queries comes into play. This innovation could foster inclusivity and user-friendly data utilization by making databases more accessible.

SQL generation refers to the process of automatically creating SQL queries based on various inputs or criteria. The objective is to distill key semantics of the natural language and crucial details, making the content more accessible and easily executable.

1.1. Structured Query Language (SQL)

Structured Query Language (SQL) serves as the lingua franca of relational database management systems (RDBMS), providing a standardized means of retrieving, manipulating, and managing data stored in structured formats.

A query is a command expressed in a structured language, typically SQL, designed to interact with databases. It serves as the conduit between human inquiry and machine logic, translating questions into executable instructions that databases understand. With a well-crafted query, users can retrieve, manipulate, and analyze data with remarkable efficiency and accuracy. The procedure is as follows:

Step-1: Define the Objective: Mention the information that is needed to be retrieved or from the database.

Step-2: Identify the Tables: Determine the tables in the database that contain the relevant data for the query.

Step-3: Select Columns: Identify which of the columns from the tables are needed for the query result.

Step-4: Specify Criteria: If the need to filter the data is required, then specify the conditions that the data must meet. This is done using the WHERE clause.

Step-5: Join Tables (if necessary): If data is needed from multiple tables,

determine how these tables are related and join them using appropriate join operations (INNER JOIN, LEFT JOIN, etc.).

Step-6: Grouping (if necessary): If there is need to group the data for aggregation, use the GROUP BY clause.

Step-7: Sorting (if necessary): If data has to be sorted in a specific order, use the ORDER BY clause.

Step-8: Limiting Results (if necessary): If there is need only for a specific number of rows from the result set, use the LIMIT clause.

Step-9: Query: Based on the above steps, SQL query is constructed using the appropriate syntax for your data requirement.

Advantages of Structured Query Language:

- i. Scalability: SQL databases are highly scalable, capable of handling large volumes of data and supporting thousands to millions of concurrent users. They can efficiently manage data growth over time without sacrificing performance.
- ii. Data Integrity: SQL databases enforce data integrity through constraints such as primary keys, foreign keys, and check constraints. These mechanisms prevent invalid data from being inserted or modified, ensuring the consistency and reliability of the database.
- iii. Data Recovery: SQL databases provide mechanisms for data backup, restoration, and recovery in case of system failures, hardware malfunctions, or human errors. Backup and restore operations can be automated and scheduled to minimize downtime.

Challenges of Structured Query Language:

- i. Complexity: SQL can become complex for managing large and complex databases. Writing efficient queries and optimizing database performance may require advanced knowledge and expertise.
- ii. Limited Support for Unstructured Data: SQL databases are designed for structured data with well-defined schemas. They may have limited support for unstructured or semi-structured data types, such as text, images, or JSON documents.

1.2. Natural Language Processing (NLP)

It is a subfield of computational linguistics that focuses on the interaction between computers and humans through natural language. Its primary objective is to enable computers to understand, interpret, and generate human language in a way that is both meaningful and useful. Some of the key aspects and components of NLP are:

- i. Preservation Tokenization: Breaking down text into smaller units such as words, phrases, or sentences. This is the first step in many NLP tasks.
- ii. Part-of-Speech Tagging: Assigning grammatical categories (e.g., noun, verb, adjective) to words in a sentence.
- iii. Named Entity Recognition: Identifying and classifying named entities (such as people, organizations, locations) within text.
- iv. Syntax and Parsing: Analyzing the grammatical structure of sentences to understand relationships between words and phrases.
- v. Semantic Analysis: Understanding the meaning of text beyond its literal interpretation, including word sense disambiguation and semantic role labeling.
- vi. Information Retrieval: Retrieving relevant information from large collections of text, often using techniques like keyword matching and relevance ranking.
- vii. Sentiment Analysis: Determining the sentiment or opinion expressed in a piece of text, often classified as positive, negative, or neutral.
- viii. Text Classification: Categorizing text documents into predefined classes or categories, such as spam detection, sentiment analysis, or topic classification.
- ix. Grammatical Accuracy: Extracted sentences are grammatically correct and coherent since they are directly taken from the source.
- x. Simplicity: Extractive methods are generally simpler to implement compared to abstractive methods, as they don't involve generating new content.

Advantages of Natural Language Processing:

- i. Multilingual Communication: NLP facilitates communication across language barriers by enabling translation and interpretation between different languages.
- ii. Efficiency: NLP automates many tasks that would otherwise require human intervention, leading to increased efficiency and productivity.
- iii. Personalization: NLP enables personalized interactions with users by

understanding their language preferences and adapting responses accordingly.

Challenges of Natural Language Processing:

- i. Ambiguity: Natural language is inherently ambiguous, making it challenging for systems to accurately interpret and process human language.
- ii. Domain Specificity: NLP models trained on general text may not perform well in specialized domains or industries with unique terminology and language conventions.
- iii. Complexity: NLP tasks such as language understanding, generation, and translation are highly complex and may require sophisticated algorithms and large datasets to achieve accurate results.

1.3. Motivation

The motivation to develop an SQL Generator arises from the palpable challenges experienced by individuals lacking technical expertise. The sheer volume of datasets, coupled with the complexity of language and semantics of the language poses a substantial hurdle to the efficient SQL generation. As a consequence, the landscape calls for a novel approach that not only accelerates the generation process but also ensures the preservation of intricate semantic details and produces executable queries.

SQL Generation, as a process, holds the potential to revolutionize the traditional SQL queries paradigm. It involves the condensation of extensive datasets into concise and informative query generation, thereby facilitating a more accessible and time-efficient comprehension of database related information. The generators can significantly improve productivity by enabling users to quickly and intuitively retrieve information from databases. Instead of spending time learning SQL or consulting database administrators, users can directly express their queries in natural language.

The existing methods of SQL Generation have witnessed the development of query generation models, owing to their simplicity and proximity to the natural language. However, these models exhibit drawbacks such as inaccurate results, resolving ambiguity, lack of executable queries, and potential oversight of critical details. The models encounter difficulties in effectively capturing important information due to the unstructured format and nuanced nature of the natural language. They often struggle with generating SQL queries for complex questions involving multiple conditions, nested queries, or aggregate functions.

1.4. Problem Definition

The barrier of learning programming languages for retrieving information from extensive databases is a significant challenge faced by many individuals, particularly those without

a technical background. Traditional methods of querying databases often require proficiency in languages like SQL, which can be daunting for newcomers and non-technical users. The current methods employed for the SQL generation exhibit notable shortcomings. Firstly, obtaining comprehensive and diverse training data that covers the full range of possible queries and database schemas can be challenging. As a result, these models [2] may struggle to accurately interpret queries or generate appropriate SQL statements, especially for less common or complex scenarios. Natural language is inherently ambiguous, with multiple valid interpretations for a given input. This ambiguity poses a challenge for generator models, leading to errors or inaccuracies in query interpretation. Resolving ambiguity accurately requires sophisticated language understanding capabilities, which may be lacking in current models. The major issue faced is that the extensive datasets [3] often involve complex query scenarios, such as joins across multiple tables, nested subqueries, and aggregations. Existing generator models struggle to handle these complex queries effectively, as they require a deep understanding of the underlying database schema and query semantics. Generating accurate SQL statements that capture the intended meaning of the natural language input can be challenging in such cases.

1.5. Problem Illustration

In the current approach, SQL generation relies on the utilization of GNN (Graph Neural Network). However, the model is constrained by table limitations, meaning it can only generate a restricted amount of accurate queries. As the number of tables grows, the comprehensive tuples of the tables increases and it leads to growth in the execution time. Unfortunately, due to the model's table constraints, it cannot generate queries that are semantically correct. Consequently, it defaults to providing an overall gist of the natural language, becoming less relevant or accurate.

Table 1.1. Table Limitation problem in Existing Approach

S.no	No of tables	No of tuples	Accuracy	Avg time of execution
1	1	240	0.875	2 ms
2	2	765	0.814	46 ms
3	5	1864	0.746	108 ms
4	8	7642	0.584	193 ms
5	17	28469	0.352	275 ms

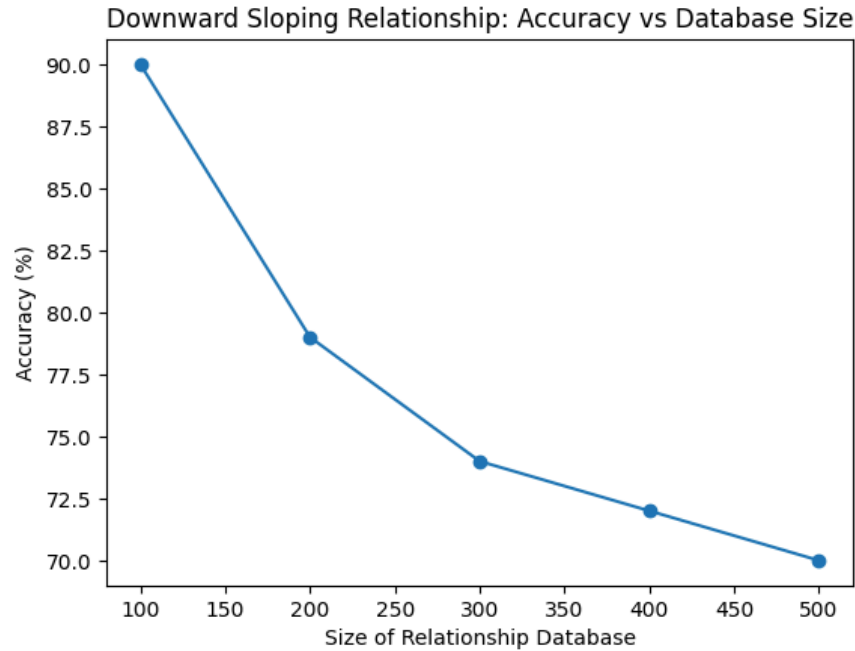


Figure 1.1. Accuracy v/s Input size

1.6. Objective Of The Project

The objective of a Natural Language to SQL Generator project is to develop a system that enables users to interact with databases using natural language queries, instead of requiring them to write SQL queries manually. By providing a user-friendly interface for interacting with databases, the project aims to democratize access to data, making it accessible to a wider audience beyond technical experts or database administrators. Natural Language to SQL generator aims to improve productivity by reducing the time and effort required to formulate and execute database queries. Users can quickly retrieve the desired information without having to learn SQL syntax or navigate complex database structures. It aims to handle complex queries involving multiple tables, joins, aggregations, and other advanced SQL constructs. The project seeks to develop models capable of understanding and generating SQL queries for a wide range of query scenarios and database schemas. Despite challenges, such as ambiguity in natural language, the project aims to develop models that can accurately interpret user queries and generate SQL statements that capture the intended meaning. By lowering the barrier to accessing and analyzing data, Natural Language to SQL generators aim to promote innovation and insights discovery. Users from various domains can explore data more freely and uncover valuable insights, leading to data being more intuitive, efficient, and accessible.

2. Literature Survey

A.Deshpande et. al. [1] proposed a model aiming to bridge the gap in SQL expertise among users interacting with databases. The methodology involves Natural Language Processing (NLP), and its component employs either pre-defined rules or trained models to map natural language to SQL constructs. However, this NLP technology relies on a well-defined and accurate database schema for accurate SQL generation. T S, Anisha et al. [2] have explored the application of deep learning in text-to-SQL conversion, aiming to automatically translate natural language questions into SQL queries within NLP. The authors conducted research, using common deep learning techniques like Sequence-to-Sequence(Seq2Seq) Models. Training and evaluation were based on datasets containing natural language queries and corresponding SQL queries. The main challenges faced in this research are handling complex queries, generalizing to unseen databases, and ensuring robustness to noise and ambiguity in natural language. K. Lahoti et al. [3] have explored the deep learning approaches for text-to-SQL semantic parsing, including the Sequence-to-Sequence Models and Structured Prediction Models. These methods are utilized for encoding-decoding language, breaking down SQL queries, and pre-training large models. However, achieving high accuracy in translating complex natural language queries into correct SQL remains challenging. A. Kate et al. [4] address the automatic translation of natural language questions into executable SQL queries. Rule-based systems are used for predefining the rules for constructing the SQL and machine learning models for training datasets of natural language queries and corresponding SQL queries to learn the translation patterns. M. Arefin et al. [5] developed a model that automatically translates natural language questions into executable SQL queries. They incorporated supervised learning for data collection, preprocessing, model selection, training, and evaluation. However, they struggled with the availability of quality datasets for training, which were crucial but limited in size and diversity. P. Parikh et al. [6] address the challenges associated with SQL by creating an application that transforms natural language questions from course instructors into SQL queries. This is achieved through the implementation of a Seq2Seq model, which generates SQL queries to retrieve relevant data displayed on a dashboard through an e-learning domain. X. Xu et al. [7]

opted for supervised learning that employs Structured prediction with Sketch-Based decoding, breaking down SQL query generation into structured decisions using sketches for syntactic correctness and reduced search space. The method faces scalability problems. T. Shi et al.[8] the incremental Text-to-SQL model generates SQL queries, employing a Seq2Seq model with attention or a structured prediction approach, enabling context-aware decisions and potential revisions during query generation. It undergoes non-deterministic Oracle training, learning from examples where multiple correct SQL queries are possible for a given input. This approach teaches the model to handle ambiguity, encouraging the production of diversity. But this approach requires a lot of training time. C. Wang et al. [9] employed an iterative decoding with execution feedback, generating SQL queries in multiple steps and interacting with the database at each stage. Error correction and refinement mechanisms detect and address potential errors in generated queries based on execution feedback, enhancing the chances of successful execution. Integrating execution feedback into the decoding process adds complexity to model design and training. Moses Visperas et. al.[10] have come up with an idea of text-to-SQL semantic parsing within Natural Language Interfaces to Databases (NLIDBs), likely conducting a comprehensive review and comparison of recent approaches. They stated the need for high accuracy in translating diverse natural language queries to SQL and the challenge of explaining the reasoning behind generated SQL queries, posing difficulties in debugging and addressing user concerns. C. Wang et. al.[11] This model aims to generate diverse and complex SQL queries, surpassing the capabilities of many text-to-SQL systems, utilizing Neural Machine Translation, Structured Prediction, and Grammar Induction methods. The approach enables the generation of intricate queries and the learning of new structures and patterns from additional examples for enhanced flexibility. M. Brockschmidt et.al.[12] developed automatically identifying and extracting SQL queries embedded within natural language text, separating them from surrounding text. In this model the analysis approaches are used, using these approaches parsing of the queries and analysis of query semantics is done to confirm alignment with the surrounding text context. N. Yaghmazadeh et. al.[13] have developed a translation of natural language questions about a database into executable SQL queries. Mapping Stage is used for establishing connections between

natural language and database structures. Sqlizer dynamically generates a structured SQL query that retrieves the information requested in the natural language question. However, models trained on specific databases might not perform well on unseen database structures. Po-Sen Huang et al.[14] methodology employs meta-learning to generate structured queries from natural language by treating each example as a unique task, creating pseudo-tasks, and using a meta-learning algorithm like Model-Agnostic Meta-Learning (MAML) for quick adaptation. J. M. Zelle et al.[15] have observed that logical inductive programming involves providing a knowledge base (KB) with rules about database query structure, encompassing both general syntactic rules (e.g., SQL syntax) and domain-specific rules. The learning process utilizes training examples, pairing database queries with correct parse tree representations, as the Inductive Logical Programming (ILP) algorithm iteratively searches for logical rules. When parsing new queries, the learned rules are applied to infer logical structures and generate parse trees for queries. It consumes a lot of time to train the model. B. Bogin et. al.[16] have come up with a methodology involving Schema Graph Construction, representing the database schema as a graph where nodes depict tables, columns, and attributes. A Graph Neural Network (GNN) is then applied to learn representations of each graph node, capturing structural and semantic information, including relationships and dependencies. In Text-to-SQL parsing, the learned schema graph representations are integrated into the model to inform query generation, guiding the process toward producing SQL queries aligned with the database structure. C.Sugandhika et.al.[17] have proposed a Heuristics Application phase that employs predefined rules to map natural language elements to SQL constructs, addressing specific patterns such as mapping nouns to tables and handling aggregations. SQL Query Generation assembles components based on heuristics, ensuring syntactic correctness and schema adherence. Heuristics might not always capture the correct intent of the user's query. A. Anisyah et. al[18] have developed a Natural language interface to bridge the gap between natural human language and structured database queries, empowering users to interact with databases without requiring technical SQL knowledge. Xiaoyu Zhang et. al.[19] have proposed the Multi-task SQL (M-SQL), leveraging a pre-trained Bidirectional Encoder Representations from Transformers (BERT) model for semantic understanding. It

integrates modules for value extraction and column matching, identifying query values and associating them with relevant table columns. The joint decoder utilizes BERT's encoded representation and outputs from extraction modules to generate the final SQL query. I.Androutsopoulos et.al.[20] have proposed an SQL that operates through three stages: understanding the query using a user-defined knowledge base, generating an SQL query in a Logical Query Language (LQL), and refining the query for errors and optimization. It poses an issue with potential performance overhead for large databases. Muhammad Shahzaib Baiget et al.[21] review the translating natural language queries into SQL compares existing frameworks based on aggregation classifier, select column pointer, and clause pointer, emphasizing the role of semantic parsing and neural algorithms in predicting these components. B. Sujatha et al.[22] Efficiently Compliant Flexible Compliant Natural language (EFFCN) interface to a database involves parsing the query to identify key elements, mapping this information to SQL clauses using rules or templates, and refining the generated SQL. Potential reliance on handcrafted rules may limit its ability to handle unseen or complex queries. Abhishek Kharade et al.[23] generate SQL queries using Natural Language Processing and review encompasses existing approaches, ranging from rule-based systems to advanced machine learning models, highlighting diverse methodologies in the field. It concludes with the performance issues and lack of accuracy in the existing methods. Adrián Bazaga et al.[24] model adopts a "polyglot" approach for translation, allowing it to generate database specific SQL for various engines. It uses a rule-based parser to extract key information. The model performance depends on the quality and size of the training data. Pwint Phyu et al. [25] Use NLP on database queries allowing users to interact with databases using plain English. This process involves understanding the user's question, interpreting the semantic meaning to match the database schema, and then generating and executing an SQL query. T. H. Y. Vuong et al.[26] In the initial phase, specific patterns are utilized to recognize components like the SELECT clause and assign them appropriate labels. In the subsequent phase, relevant keywords and entities are extracted for each labeled component, resolving any ambiguities along the way. Extracted values are then inserted into the corresponding slots within the SQL clauses. Finally, these filled slots and clauses are merged to produce the comprehensive SQL query. G. Rao et al.[27]

translation to SQL involves establishing formal grammar for sentence structures and word combinations forming semantic parser interpretation. This semantic representation is then matched to the database schema, aligning entities with tables and columns. This process generates a valid SQL query, enabling retrieval of desired data from the database.

P. Pasupat et al.[28] logical form grammar for NLP-based database queries focuses on constructing logical forms that match user queries with database tables, through a process of parsing the query, checking types against the table schema, finding the best matching logical form, and generating an answer.

A. Pagrut et al.[29] automated SQL query generator that employs NLP to understand user input, matching it with database elements and discerning conditions and operators. It then constructs a valid SQL query, it validates the query, allowing users to review and adjust it as necessary, ultimately executing it on the database and presenting the results. It often requires customization for specific domains and database schemas.

R. Kumar et al.[30] The method involves analyzing the user's natural language query through tokenization and morphological analysis, identifying word forms and meaningful phrases. A pattern library links these language structures to the SQL constructs via a pattern matching algorithm, to match for elements like tables, columns, and operators. The resulting SQL query is executed on the database.

V. Zhong et al.[31] a neural network-based model called Seq2SQL, which consists of an encoder-decoder architecture. Seq2SQL employs a sequence-to-sequence model with attention mechanisms. The encoder is a bidirectional LSTM (Long Short-Term Memory) network, which encodes the input question into a fixed-size vector representation. The decoder is also an LSTM-based network, which generates the SQL query token by token, conditioned on the encoder's output and the tokens generated. The encoder processes the input question (in natural language), while the decoder generates the corresponding SQL query. To handle the structured nature of SQL, the decoder outputs tokens representing the different components of the query (such as the SELECT, WHERE, and JOIN clauses).

T. Yu et al.[32] introduce the SPIDER dataset, which consists of over 10,000 human-annotated queries across 200 databases in diverse domains. The annotation process involves multiple steps, including query generation, query simplification, SQL schema identification, and SQL query annotation. Each query is annotated with the corresponding SQL code. Human annotators were involved at each step to ensure

high-quality annotations. The dataset also includes detailed annotations for query structure and linguistic phenomena. The SPIDER dataset poses several challenges for text-to-SQL models, including complex queries with nested clauses, aggregation functions, and joins across the multiple tables. J. Guo et al.[33] a novel approach that involves using an intermediate representation (IR) to bridge the gap between natural language and SQL. The IR serves as an abstract semantic representation that captures the meaning of the input question in a database-agnostic manner. By utilizing this intermediate step, the model aims to generalize better across different database schemas and domains. To handle databases from different domains, the model is trained on a diverse dataset containing examples from various domains. This helps the model learn to generalize across different schemas and linguistic patterns. The evaluation assesses the model's ability to accurately translate complex natural language questions into SQL queries across different domains and database schemas. H. Long et al.[34] BERT, a pre-trained transformer-based model known for its effectiveness in various natural language processing tasks has been used. To improve the understanding of the relationships between questions and database tables, the method incorporates techniques to enhance the representation of question-table content. This helps the model capture more nuanced semantic information crucial for generating accurate SQL queries. The method employs template filling strategies to guide the SQL query generation process. Templates are predefined structures that capture common query patterns. By filling in these templates based on the context of the question and database schema, the model can generate SQL queries that are more effective. S. S. Badhya et al.[35] converting natural language queries into structured SQL queries tailored for descriptive columns using Elasticsearch. By leveraging elasticsearch's capabilities in handling textual data, the proposed method enables effective querying of databases containing descriptive columns. Natural language queries are parsed to identify relevant keywords and entities. These keywords and entities are then used to construct Elasticsearch queries to search for matches within the descriptive columns. The generated SQL queries incorporate conditions and filters to match the user's intent expressed in the natural language query. Richard Socher et al.[36] utilizes an editing-based approach to generate SQL queries. Instead of generating the query from scratch, the model starts with an initial query

template and then edits it based on the context provided in the question. This allows the model to leverage existing query templates and adapt them to the specific context of each question. It is designed to generalize across different domains by training on a diverse dataset containing examples from various domains. This enables the model to learn patterns and relationships that are common across different domains, improving its ability to generate accurate queries for context-dependent questions. Oleksandr Polozov et al.[37] enhances the encoding of database schemas by incorporating relation-aware features. This allows the model to better understand the relationships between different database tables and columns. Specifically, relation-aware features are used to represent the relationships between foreign keys and their corresponding primary keys, as well as hierarchical relationships between tables. The method even includes a linking mechanism that maps natural language utterances to specific database schema elements, such as tables and columns. This linking is crucial for accurately generating SQL queries based on the input questions. The linking mechanism utilizes both schema information and context from the input question to identify relevant schema elements. Matthew Purver et al.[38] it takes as input natural language specifications provided by users, which describe the desired information retrieval tasks in plain language. It uses semantic parsing techniques to analyze and understand the natural language specifications. By parsing the natural language input, the model identifies key components such as entities, attributes, conditions, and operations. These specifications can range from simple queries to more complex ones involving multiple tables and conditions. Based on the parsed information, the method generates the corresponding SQL queries that fulfill the specified tasks. The generated SQL queries are designed to be syntactically correct and semantically aligned with the user's intentions. R. Zhong et al.[39] involves creating the distilled test suites that contain semantically equivalent queries for each original query in the evaluation dataset. These distilled test suites are constructed using various techniques, including paraphrasing, query reformulation, and data augmentation, to ensure that they capture a wide range of semantic variations. The distilled test suites are used to evaluate the semantic equivalence between generated SQL queries and ground truth queries.

Table 2.1. Comparison of Existing Strategies

S.no	Strategies	Advantages	Disadvantages
1	TypeSQL: Knowledge-based Type-Aware Neural Text-to-SQL Generation	TypeSQL generates SQL queries faster than some comparable models.	Lack of accuracy when dealing with the massive datasets
2	Robust Text-to-SQL Generation with Execution-Guided Decoding	Execution guidance significantly improves both syntactical and execution accuracy of the models.	Executing the partially generated queries incurs additional computational overhead, potentially impacting real-world performance.
3	Text to SQL Query Conversion Using Deep Learning: A Comparative Analysis	Outperforms rule-based and statistical approaches in terms of generating accurate SQL queries.	The models may not handle all possible SQL features, such as subqueries and complex data types.
4	Querylizer: An Interactive Platform for Database Design and Text to SQL Conversion	The interactive approach helps minimize errors in schema design and query formulation compared to manual SQL coding.	The accuracy of generated SQL code relies heavily on the clarity and completeness of the user-provided text descriptions. Misinterpretation of text could lead to errors.
5	M-SQL: Multi-Task Representation Learning for Single-Table Text2SQL Generation	Multi-task learning stabilizes training and makes the model less susceptible to specific query patterns or schema complexities.	M-SQL focuses on single-table scenarios and might not generalize well to complex queries involving multiple tables or joins.

3. Natural Language To SQL Generator

The proposed approach is aiming to get a notable improvement in the semantic accuracy, which indicates that the generated query accurately captures the content of the entire input. Additionally, it aims to overcome the limitations of working with the extensive databases and generating of the multi-column queries. The procedure consists of two independent stages: the first stage is fine-tuning the model, and the second stage is using the fine-tuned model to generate queries.

3.1. Fine-tuning the model

Fine-tuning improves the performance of models that are available via the API tailoring it to specific tasks or domains, surpassing the outcomes achieved through prompting methods. It enables training on larger datasets, surpassing the limitations imposed by prompt-based approaches. Succinct prompts save tokens, optimizing efficiency, while reducing latency in query processing. Below flowchart in Figure 3.1. gives an overall summary of fine-tuning.

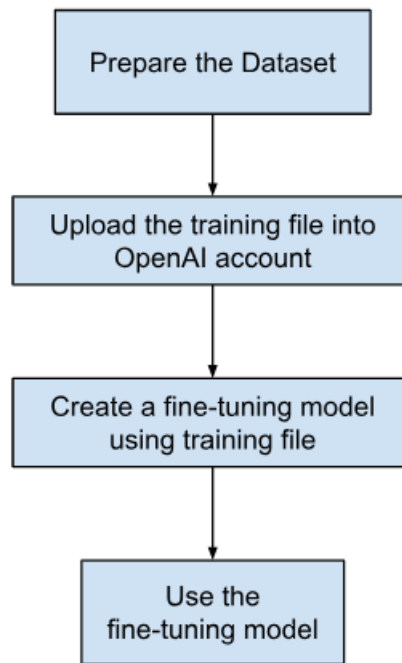


Figure 3.1. Overview of fine-tuning procedure.

The following steps are included in the fine-tuning process:

3.1.1 Prepare and upload training data

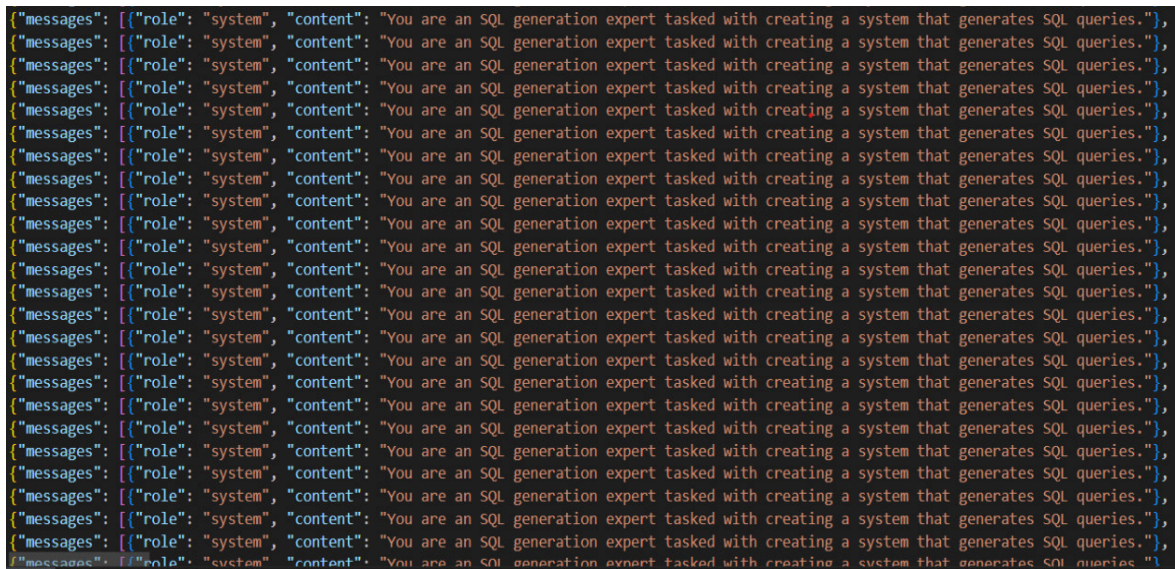
The dataset must have the same conversational structure as the Chat Completions API. This implies that it ought to consist of a series of messages, with each message having its own content and a role (such as the speaker). This can only occur if the dataset is

converted to a JSON lines file type.

For example:

```
{"messages": [{"role": "system", "content": "You are an SQL generation expert  
tasked with creating a system that generates SQL queries. "},  
  
{"role": "user", "content": "schema and question"},  
  
{"role": "assistant", "content": "SQL query"}]}
```

The assistant will get instructions from the system regarding how to handle the data that the user has provided it with. We created the dataset in JSON lines format as shown in Figure 3.1.

A screenshot of a text file containing JSON lines. Each line represents a single message in a conversation. The messages are grouped into three categories: 'system', 'user', and 'assistant'. The 'system' messages are all identical, stating 'You are an SQL generation expert tasked with creating a system that generates SQL queries.' The 'user' messages are all identical, stating 'schema and question'. The 'assistant' messages are all identical, stating 'SQL query'. The lines are separated by commas and newlines, forming a continuous stream of JSON objects.

Screenshot 3.2. JSON lines dataset

After preparing the training data it must then be uploaded via the Files API. This makes it possible to use the data for fine-tuning tasks efficiently. Apply the subsequent steps:

Step-1: Import the OpenAI module.

Step-2: Set up an OpenAI client instance for communication.

Step-3: Use binary read mode to access the JSON Lines dataset.

Step-4: Upload the opened file to a new file using the OpenAI client.

Step-5: Set the purpose of the file as "fine-tune."

3.1.2 Create a Fine Tune Model

After uploading the file, the next action is to initiate a fine-tuning job. To achieve this, use the OpenAI SDK to create a fine-tuning job:

Step-1: Set up an OpenAI client instance for communication.

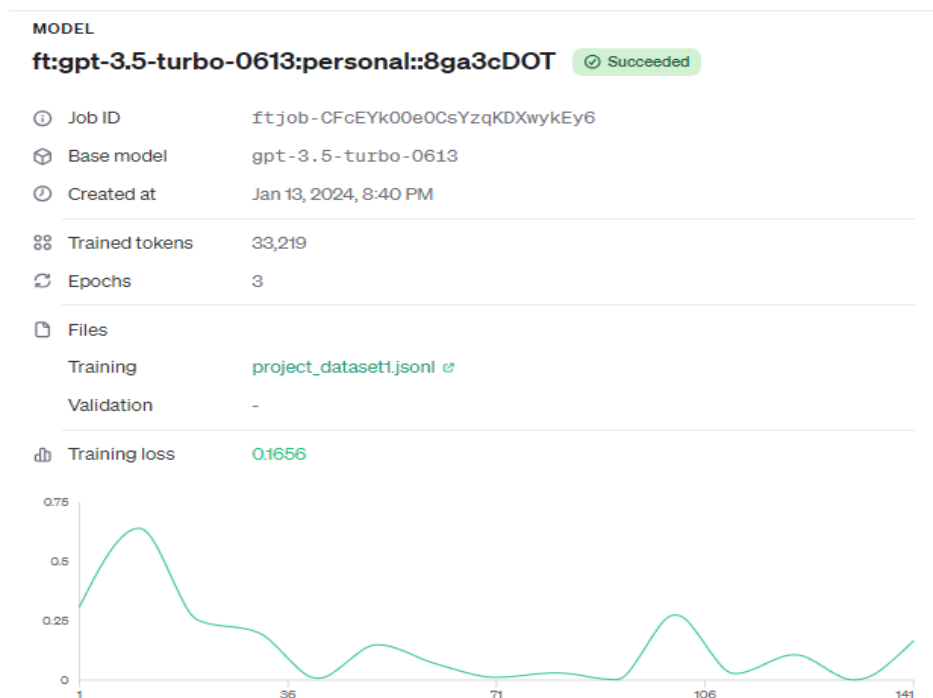
Step-2: Create a fine-tuning job using the OpenAI client.

Step-3: Enter the training file ID (such as "file-abc123") that was returned when the training file was uploaded to the OpenAI API as its identifier.

Step-4: Select "gpt-3.5-turbo" as the model to be fine-tuned.

When initiating a fine-tuning job, the completion time can vary due to queue dynamics and training factors such as model and dataset size. Once the training concludes, users receive a confirmation email.

Upon successful completion, the fine-tuned model field in the job details will be populated with the name of the specific model and even the fine-tuned model will be reflected in your OpenAI as shown below in Figure 3.3.



Screenshot 3.3. Details of fine-tuned model on OpenAI account

The overall algorithm for fine tuning:-

Step 1: Import pandas

Step 2: Upload the csv dataset

Step 3: Encode the dataset with 'cp1252'

Step 4: Create a new jsonl file named 'training.jsonl' and open it in writing mode

Step 5: Create a function CREATE_DATASET which takes a schema, question and its corresponding query as the input parameters.

Step 6: The function creates a message as shown in the above example and returns it.

Step 7: Iterate through the CSV file and send the schema, question and query one by one to CREATE_DATASET.

Step 8: While iterating write the returned message (every time in a new line) from CREATE_DATASET in training.jsonl.

Step 9: From 'openai' library import 'OpenAI' class

Step 10: Create an instance of the OpenAI class as 'client' with an 'api_key' parameter set as user API KEY , which will be used to communicate with the OpenAI API.

Step 11: Upload the training file using 'file.create()' method under 'OpenAI' and set its 'file' parameter as open 'training.jsonl' file in read binary mode and set the 'purpose' parameter as 'fine-tune'. A file id will be generated

Step 12: Fine tune the model using 'fine_tuning.jobs.create()' method under OpenAI class and set its parameters. Set 'training_file' as the file id generated after uploading the training file, set the 'model' as the 'gpt-3.5-turbo'. This will create a fine-tuning job id.

Step 13: Retrieve the fine-tuned model status using fine_tuning.job.retrieve() method under OpenAI. The model name will be generated.

3.2. Generating summaries using fine-tuned model

In our generation process, the input produced by the user is processed by our LLM model. Based on the input the LLM model generates the necessary query. Once the query

is generated, it is executed against the database provided by the user to retrieve the information required by the user.

3.2.1 Install python libraries

During the initial phase of workflow, we proceed with the installation of three Python libraries:

- i) OpenAI:* Incorporated to utilize the robust language models provided by OpenAI for the purpose of generating queries.
- ii) LangChain:* Crucial for the efficient transformation of the natural language to SQL query generation.
- iii) Psycopg2:* It is essential for establishing a connection with the database server and to access the database for retrieval of the data required.
- iv) Chroma:* It manages and creates vector representation of the CSV files.

3.2.2 Launch the OpenAI LLM

To Utilize advanced language models for SQL query generation, follow the describe process:

Step 1: Import the required libraries from the langchain.chat_models module, including OpenAI for interacting with the OpenAI API and ChatOpenAI.

Step 2: Define the API key for authentication with the OpenAI API.

Step 3: Build an instance of the ChatOpenAI class with parameters such as the desired model, temperature, maximum tokens, and API key.

3.2.3 Obtain details of the table

Retrieving basic details about tables in the database. The function executes a SQL query against the database to retrieve basic details about tables in the schema. The query selects three columns from the 'information_schema.columns' view: 'table_name', 'column_name' and 'data_type'. The SQL query is executed using the 'cursor.execute()' method with the cursor object passed to the function as an argument. The 'fetchall()' method retrieves all the rows returned by the query. The function returns these fetched rows, which represent basic details about tables in the schema.

3.2.4 Retrieve information about the foreign key

Retrieve information about foreign keys in the database. A function constructs a SQL query stored in the variable 'query_for_foreign_keys'. This query retrieves information about foreign

keys from the system catalog tables 'pg_constraint' and 'pg_attribute'. The query is executed using the 'cursor.execute()' method with the cursor object passed to the function as an argument. 'cursor.fetchall()' retrieves all the rows returned by the query. The function returns the fetched rows, which represent information about foreign keys in the database.

3.2.5 Creating vector representations

The purpose of creating vector representations is to encode and represent data from a Chroma vector database in a format that is suitable for computational analysis and processing.

Step-1: *Initializing the CSVLoader object* - This step involves setting up a tool or library capable of loading data from CSV files, which are commonly used for storing tabular data. This initializes the process of accessing the Chroma vector data stored in CSV format.

Step-2: *Loading the data from the CSV file*- In this step, the actual data stored in the CSV file containing Chroma vector information is read and imported into memory.

Step-3: *Chroma vector database is to be created for vector representation*- This is the crucial step where the loaded data is processed and organized into a structured database format, specifically tailored for representing Chroma vectors.

3.2.6 Save the details related to the database schema

It is to ensure the structured storage and management of database metadata, particularly pertaining to table information and foreign key relationships.

Step-1: *Generating a unique identifier* - This step involves creating a unique identifier for the database schema, which helps in tracking and referencing the schema throughout the database management process. It ensures consistency and reliability in schema management tasks.

Step-2: *Establishing the database connection* - Establishing a connection to the database enables access to its metadata, such as table names, column details, and foreign key

relationships. This step is essential for retrieving the schema information required for subsequent processing.

Step-3: *Storing table information in a data frame* - The information retrieved about the database tables is organized into a structured format, typically a data frame, for easy manipulation and analysis. The data frame is then saved as a CSV file.

Step-4: *Storing foreign key details in a data frame* - Foreign key relationships, which define connections between different tables in the database, are extracted and structured into a data frame. Saving this dataframe as a CSV file facilitates understanding and management of database relationships.

Step-5: *Developing vector representations* - Once the database schema details are saved, vector representations can be developed based on the schema's structural information. This involves encoding the schema elements into numerical vectors, enabling further analysis or processing using computational techniques.

3.2.7 Generate SQL query

Step-1: Process the input provided by the user.

Step-2: *Loading table information from CSV file* - Table information stored in a CSV file, likely generated from database schema details, is loaded into memory. This includes details such as table names, column names, data types, and possibly foreign key relationships.

Step 3: *Performing Chroma vector search* - Chroma vector search is conducted to identify and retrieve the tables relevant to the user's input criteria. This step involves querying the database schema or metadata to locate tables associated with Chroma vector data or related attributes.

Step-4: *Processing relevant tables* - Once the relevant tables are identified, they are processed from the data frame. This may involve filtering, joining, or aggregating data from multiple tables to meet the user's query requirements.

Step-5: *Processing relevant foreign keys* - Foreign key relationships associated with the relevant tables are processed according to the user input. This ensures that data retrieval is performed in a manner that maintains referential integrity and consistency across related tables.

Step 6: Interact with the large language models and generate the query.

3.2.8 Establishing database connection

The connection with the database is developed to access the database and retrieve the information.

Step-1: *Connection development with URI* - The first step involves establishing a connection with the database using the URI provided by the user. This URI typically contains information such as the database type (e.g., MySQL, PostgreSQL, MongoDB), host address, port number, database name, and authentication credentials. Establishing this connection allows applications or users to interact with the database and perform operations such as querying, updating, and retrieving data.

Step-2: *Saving database details* - Once the database connection is established, details related to the database are saved for reference and management purposes.

3.2.9 Executing the provided SQL query

The query is executed against the database to retrieve the data that is required by the user.

Step-1: *Connect to the database* - The initial step involves connecting to the database server. This connection allows the application or user to communicate with the database and submit queries for execution. It typically includes providing authentication credentials, such as a username and password, to access the database.

Step-2: *Parsing through the provided query* - Once the connection is established, the provided SQL query is parsed to identify its individual components, such as the type of operation (e.g., SELECT, INSERT, UPDATE, DELETE), table names, column names, conditions, and any other parameters specified in the query. Parsing ensures that the query is correctly interpreted and formatted for execution.

Step-3: *Executing the parsed query against the database server* - The parsed and extracted query is then executed against the database server. This involves sending the query to the DBMS, which processes it and retrieves the data

Step 4: *Displaying the response of the executed query* - Finally, the response generated by the execution of the query against the database server is displayed to the user.

The overall algorithm of sql generation:

Step 1: Install OpenAI, LangChain, and Psycopg2

Step 2: Import the OpenAI library and assign the user's API key to the '`API_KEY`' variable.

Step 3: From the 'OpenAI' library, import the 'ChatOpenAI' class.

Step 4: Initialize an instance of the ChatOpenAI class as 'llm' with parameters. Set 'temperature' to 0 to keep the output more focused and less random, set 'max_tokens' to 1000 to process the input within that limit, set 'openai_api_key' to API_KEY to access the OpenAI API, and set 'model' to the model name obtained after fine-tuning.

Step 5: Define a function that takes the 'cursor' as its input parameter. The function retrieves basic information about tables and their columns from a PostgreSQL database using SQL queries and the results are fetched using the cursor's 'fetchall' method .

Step 6: Create a function that takes 'cursor' as an input parameter essentially retrieves information about foreign key constraints in the database and returns it in a structured format, typically as a list of tuples or a similar data structure. It fetches the results using the cursor's 'fetchall' method.

Step 7: Define a function 'create_vectors' that takes two arguments: 'filename' (the path to the CSV file) and 'persist_directory' (the directory where the generated vectors will be stored).

Step 8: Initialize a CSVLoader object loader with the specified file path (filename) and encoding (utf8). Use the load method of the CSVLoader object to load the data from the CSV file. The loaded data is stored in the variable data.

Step 9: Import the 'Chroma' class from the vectorstores module from the long-chain package for the vector representation of the CSV files that are created by using chroma.

Step 10: Initialize a 'Chroma' object 'vectoradb'. Use the 'from_documents' method of the Chroma object to create vectors from the loaded data (data). It takes the loaded data as input, along with some additional parameters such as embedding (presumably a method for creating embeddings) and persist_directory

Step 11: Import the 'uuid4' class from the uuid package to generate a unique ID for every CSV file and vector representation.

Step 12: Import the pandas package to store the details about the database in the data frame.

Step 13: Retrieve the information of the tables in a dataframe and save the dataframe to a CSV file with a filename containing the unique ID.

Step 14: Create a Pandas DataFrame (df) from the retrieved foreign key details and save the dataframe to a CSV file with a filename containing the unique ID.

Step 15: Call the function 'create_vectors' to create vectors for the tables using the CSV file created.

Step 16: Import the 'ChatPromptTemplate' class from the langchain package.

Step 17: Import the 'HumanMessagePromptTemplate' class from the chat module of the langchain package, for processing the user input.

Step 18: Import the 'SystemMessagePromptTemplate' class from the chat module of the langchain package, for generating the responses to the user input.

Step 19: Define the function 'generate_template_for_sql' that takes four arguments: query, relevant_tables, table_info, and foreign_key_info, to generate a template for assisting in writing SQL queries based on provided parameters.

Step 20: Construct a 'function get_the_output_from_llm' for generating a query that takes three arguments: query, unique_id, and db_uri as its parameters and stores the result in an 'answer.content' variable.

Step 21: Import the 'Psycopg2' package to establish a connection with the database server.

Step 22: Construct an execute query method that takes the generated query and database uri as its parameters for executing the query on the provided database server. The result obtained is stored in the 'result' variable.

3.3. Illustration-SQL Generation

The following steps provide a view of how the query will be generated throughout the process.

- i. *Input provision:-* The question and the database is provided by the user as an input to gain the necessary data.

- ii. *LLM*:- The LLM model developed is used to process the question and capture the underlying semantics. It understands the intent of the question to generate the appropriate query.
- iii. *Query*:- The query is generated according to the input provided by the user, through the help of the LLM model.
- iv. *Database Output*:- The query that is generated according to the question is executed against the database to retrieve the required data.

Example on how sql generation works:-

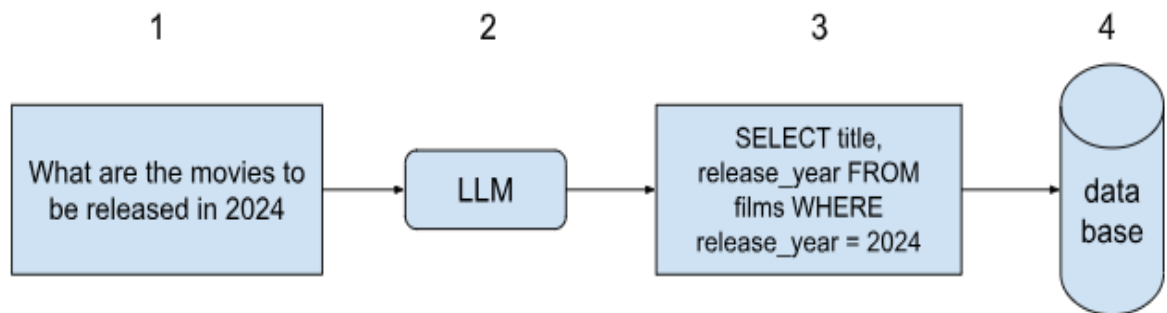


Figure 3.4. SQL Generation Example

In the above example the input provided is about the 'movies releasing in 2024', this question is then processed by the LLM model. The LLM model evaluates and captures all the semantics. Then the LLM model generates a query consisting of all the necessary conditions and aggregates that can derive the needed results. The generated query is then executed against the database that obtains all the 'movies that are to be released in the year 2024'.

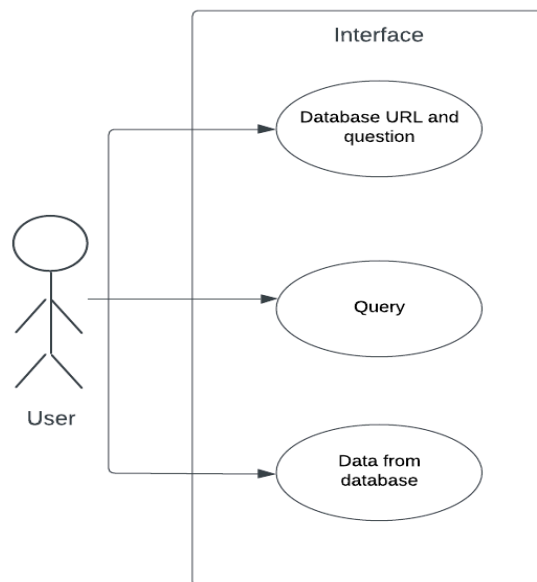
4. Design

4.1. Introduction

Unified Modeling Language (UML) diagrams are visual representations used to model the structure and behavior of software systems. UML diagrams provide a standardized way to communicate system architecture, design, and functionality among stakeholders, including developers, designers, project managers, and clients. With a wide range of diagram types, including class diagrams, use case diagrams, sequence diagrams, activity diagrams, and more, UML offers a comprehensive toolkit for capturing different aspects of software systems. Each diagram type serves a specific purpose, allowing stakeholders to focus on different facets of the system, such as static structure, dynamic behavior, interactions, and processes. By using UML diagrams, stakeholders can effectively communicate, analyze, and understand complex software systems, aiding in requirements elicitation, system design, implementation, and maintenance phases of software development projects. UML diagrams serve as a common language for expressing the system concepts and the relationships, fostering collaboration, facilitating the decision-making, and ultimately improving the quality and success of software projects.

4.2. Use case diagram

A use case diagram is a graphical representation in Unified Modeling Language (UML) that illustrates the functionalities or actions a system offers to its users (actors). Acting as a high-level overview of system behavior from a user's perspective, it helps in understanding the system's requirements and interactions.

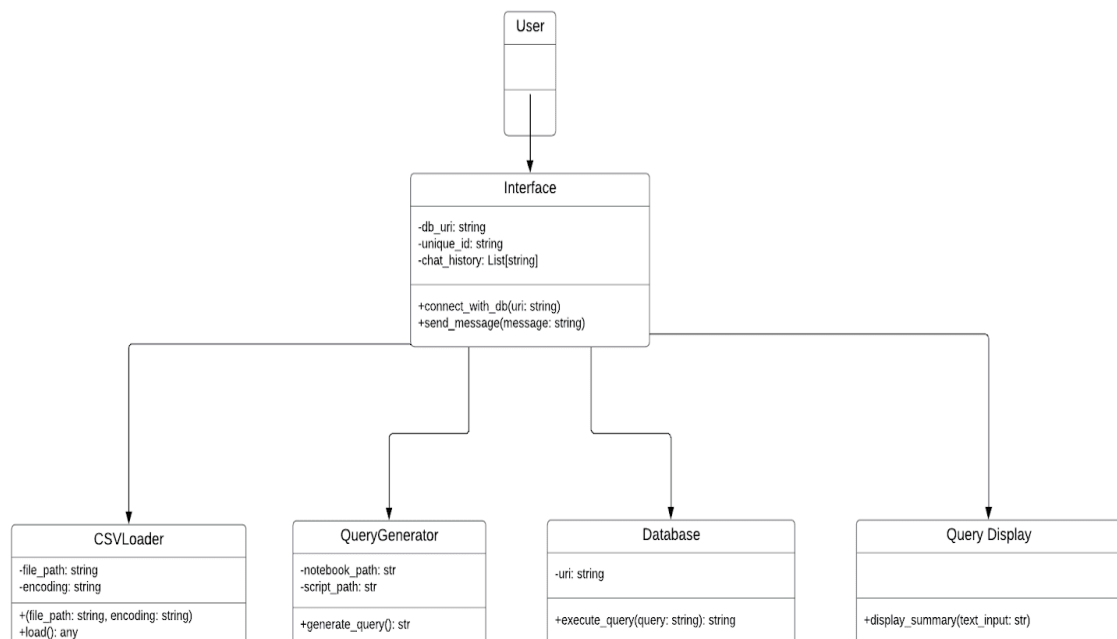


4.1. Use case diagram

In the context of a Streamlit application for generating SQL queries, the use case diagram showcases key functionalities that enable users to interact with the system seamlessly. Firstly, the "Database URI and question" use case signifies the user's ability to give a database uri and question to the application. Following this, the "Query" use case represents the action of generating a query of the given question. Once the summary is generated, the "Execute query" use case depicts how the system executes the query on the database to the user, facilitating easy comprehension. Lastly, the "Data from the database" use case allows the user to obtain data from the database for future reference or analysis. Together, these use cases delineate the core interactions between the user and the Streamlit application, outlining the primary functionalities and user-driven actions that define the system's behavior.

4.3. Class diagram

A class diagram is a fundamental type of Unified Modeling Language (UML) diagram used in software engineering to visualize the structure of a system by representing its classes, attributes, operations, and relationships. Each class is depicted as a box with three compartments: the top compartment contains the class name, the middle compartment lists the attributes or properties of the class, and the bottom compartment displays the operations or methods that the class can perform. The relationships between classes, such as associations, dependencies, generalizations, and aggregations are represented by lines connecting the classes. These relationships provide insight into how classes collaborate and interact within the system.

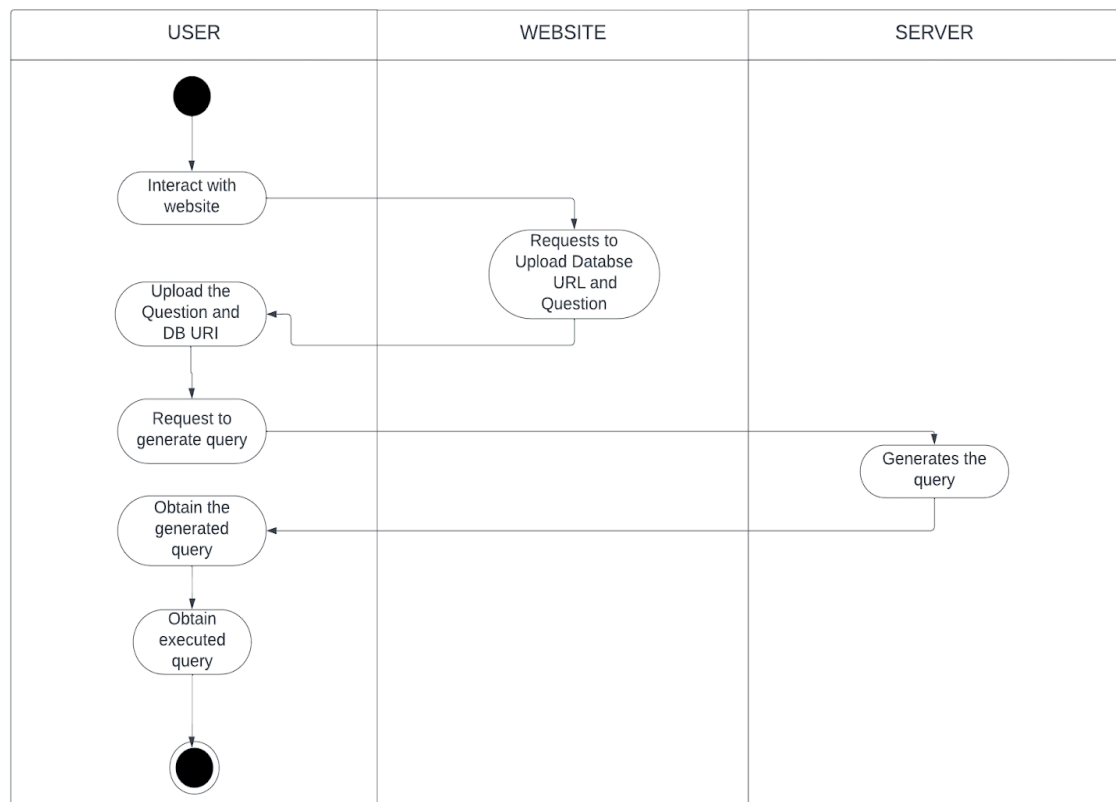


4.2. Class diagram

In the provided class diagram, several classes are depicted, each representing a distinct component of the system. For example, the 'StreamlitApp' class encapsulates the main functionalities of the Streamlit application, including text input/output and code execution. Other classes such as 'CSVLoader', 'QueryGenerator', 'Database', and 'Query Display' represent specific functionalities within the system, such as loading CSV files, generating Queries, executing queries on database, and displaying the executed queries, respectively. The arrows between classes indicate relationships, demonstrating how these components interact with each other. Overall, the class diagram provides a comprehensive overview of the system's structure, helping stakeholders understand its components, functionalities, and relationships, thereby facilitating effective communication, design, and development of the software system.

4.4. Activity Diagram

An activity diagram is a type of Unified Modeling Language (UML) diagram used to visually represent the flow of activities or actions within a system. It provides a structured way to illustrate the sequence of steps and decision points involved in a process.

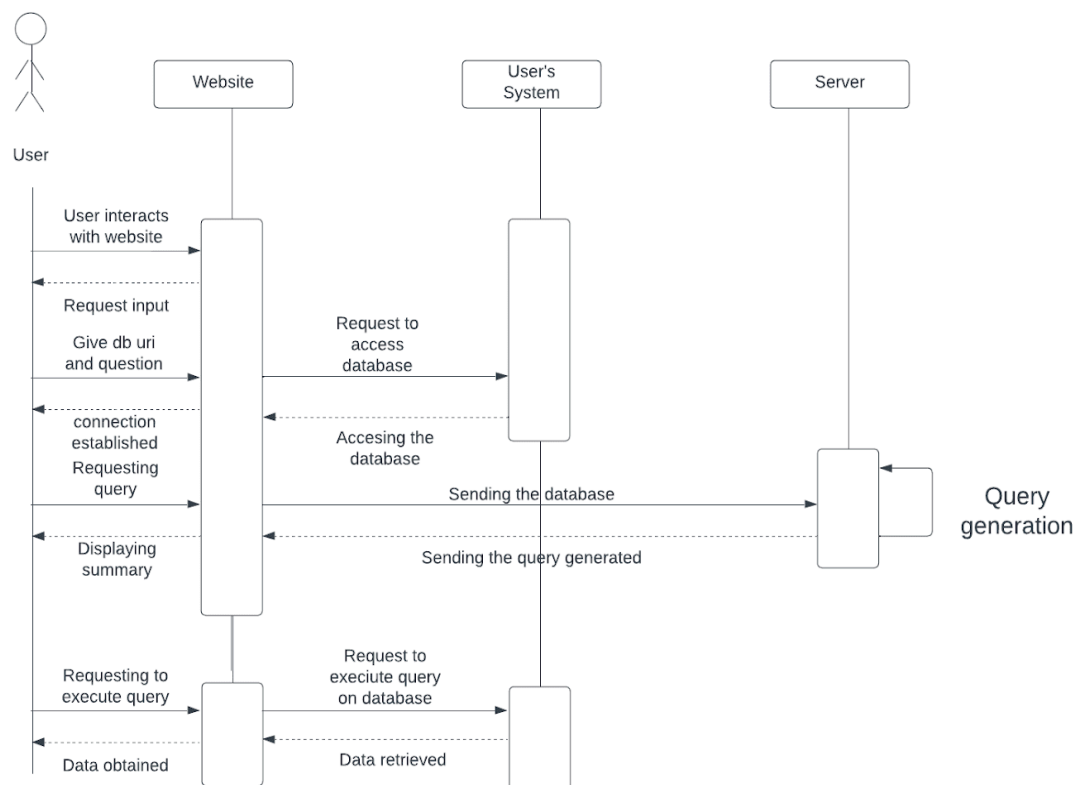


4.3. Activity diagram

In the given activity diagram, the process flow of a Streamlit application for generating SQL queries is depicted. The diagram begins with the user giving input in the form of a database uri and question. The application then checks if the input has been successfully given. If so, it proceeds to read the database path and verifies if the database exists. Upon confirmation, the application executes a snippet to generate a query, followed by displaying the generated query to the user. Subsequently, the application executes the query on the database for the user to retrieve the data from the database. However, if the input is not in the required format or the database does not exist, appropriate error messages are displayed, and the process is terminated. This activity diagram effectively outlines the sequential steps and decision points involved in the application's workflow, guiding users through the process of uploading, generating, displaying, and executing the generated queries.

4.5. Sequence diagram

A sequence diagram is a type of Unified Modeling Language (UML) diagram used to visualize interactions between objects or components in a system over time. It illustrates the flow of messages or method calls between these objects, providing a detailed view of the sequence of actions during the execution of a particular scenario.



4.4. Sequence diagram

In the provided sequence diagram, the interaction between the user and the Streamlit application for generating SQL queries is depicted. The sequence starts with the user providing a database uri and question to the application. Upon receiving the input request, the Streamlit application is activated, and it sequentially performs several actions, including reading the database path, executing a snippet to generate a query, and displaying the generated summary. Finally, the application executes the generated query on the database, after which it retrieves the data from the database. Once the data obtained is displayed to the user, the Streamlit application is deactivated, indicating the end of the interaction sequence. This sequence diagram effectively captures the chronological flow of interactions between the user and the Streamlit application, illustrating how the system processes the user's actions to generate and process the query.

5. Implementation

Program file is sql.py consisting of the code for generating queries and creating an user interface.

Input: Question and Database URL

Output: Query

5.1. Functionality

5.1.1. Getting basic table details

This function retrieves basic information about tables in a PostgreSQL database. The 'cursor' parameter represents the cursor object used to execute SQL queries in the database. The function executes an SQL query against the database to fetch basic details about tables. It selects information from the 'information_schema.columns view', which contains metadata about columns in tables. It retrieves details such as the table name, column name, and data type for each column. Once the function is called with a cursor connected to a PostgreSQL database, we will receive a list of tuples containing details about tables and their columns. The result of the query is fetched using the cursor. The function returns the list of tuples containing table details, where each tuple represents a row of information with elements '(table_name, column_name, data_type)'.

5.1.2. Foreign keys

This function retrieves information about foreign keys in a PostgreSQL database. The 'cursor' parameter represents the cursor object used to execute SQL queries in the database. The function executes an SQL query against the database to fetch details about foreign keys. It selects information from the pg_constraint system catalog table, which contains details about constraints, including foreign keys. Once this function is called with a cursor connected to a PostgreSQL database, you'll get a list of tuples containing details about foreign keys in the database. The result of the query is fetched using the cursor. The function returns the list of tuples containing foreign key details, where each tuple represents a row of information with elements '(table_name, foreign_key, foreign_key_details, referred_table, referred_columns)'. This information can be used to understand the relationships between tables, enforce referential integrity, or generate SQL queries involving joins.

5.1.3. Folder Creation

If the folder doesn't exist, it creates the folder using 'os.makedirs(folder_name)'. After creating the folder, it prints a message indicating that the folder has been created, if the folder already exists, it prints a message indicating that the folder already exists. This

function ensures that the required folders ('csvs' and 'vectors') exist in the current directory. If any of these folders are missing, it creates them.

5.1.4. CSV Files and Vector representations

This function is used for loading data from CSV files, processing it, and generating vector representations. The CSV files likely contain text data that needs to be converted into numerical vectors. Techniques such as word embeddings or document embeddings are commonly used to create vector representations of text data. These vector representations capture semantic information about the text, such as word meanings or document context. Once the vector representations are created, they can be used as input features for the model. It loads data from CSV files, creates vector representations using an embedding model (OpenAIEmbeddings class), and saves the resulting vectors for further processing or analysis.

5.1.5. Query Generation

The function is used for generating SQL queries based on user-provided natural language input. Users provide natural language questions about the database. The system needs to understand the user's query and extract relevant information from it. Based on the understanding of the user's input, the system generates corresponding SQL queries to retrieve the requested information from the database. The generated SQL queries should accurately reflect the user's intent and retrieve the relevant data from the database tables. The generated queries may involve selecting specific columns, filtering rows based on conditions, joining tables, or aggregating data. Once the SQL query is generated, it is executed against the database to retrieve the desired information. The retrieved data is then presented to the user, typically in a format that is understandable and interpretable.

5.1.6. Database connection

This function establishes a connection with the database using the provided URI. The user is prompted to enter the URI (Uniform Resource Identifier) of the database. The URI typically contains information such as the username, password, host, port, and database name required to establish a connection. It saves the URI and a unique identifier (for tracking purposes) in the Streamlit session state. It calls the 'save_db_details' function to save details about the database (such as table names, column names, and foreign key information) to CSV files. Finally, it returns a message indicating that the connection to the database has been established. Once the connection is established, the user can interact with the database through the chat interface.

5.2. Attributes:

- i. API_KEY:** It is a unique identifier or token that is used to authenticate and authorize access to OpenAI API.
- ii. temperature:** The temperature attribute controls the randomness of the generated text. A higher temperature value leads to more randomness and diversity in the generated responses, while a lower value results in more conservative and predictable responses. Here, it's set to 0, indicating that the responses will be deterministic rather than varied.
- iii. max_tokens:** This attribute specifies the maximum number of tokens (basic units of text) allowed in the generated response. Setting a limit helps manage the length of responses and prevents excessively long outputs. Here, it's set to 1000 tokens.
- iv. openai_api_key:** This attribute represents the API key required to authenticate and access the OpenAI API. The API key serves as a form of authentication and authorization to use OpenAI's services. It's typically obtained by registering with OpenAI and generating a unique key associated with the user's account. In this code, API_KEY is a placeholder for the actual API key value.
- v. model:** This attribute specifies the model to be used for generating responses. It determines the underlying architecture and parameters of the language model. Here, the model being used is 'ft:gpt-3.5-turbo-0613:personal::8f3ZRSq', which appears to be a fine-tuned version of the GPT-3.5 model. The specifics of the fine-tuning process and the particular characteristics of this model are indicated by the string.
- vi. df:** This attribute is a Pandas Dataframe. It is intended to store tabular data like 'column_name', 'table_name', 'data_type' and it also stores details about the foreign key.
- vii. folders_to_create:** This is used for ensuring the existence of necessary directories for file storage, which is crucial for saving CSV files and vectors. This is a list containing

folder names. It's used in a loop to check if these folders exist and create them if they don't. This attribute ensures that necessary directories for storing CSV files and vectors are available.

viii. db_uri: This attribute is a string variable that holds the Uniform Resource Identifier (URI) for connecting to the database. It typically contains information such as the database driver, host, port, database name, and authentication credentials. It is responsible for establishing connections with the database and executing SQL queries.

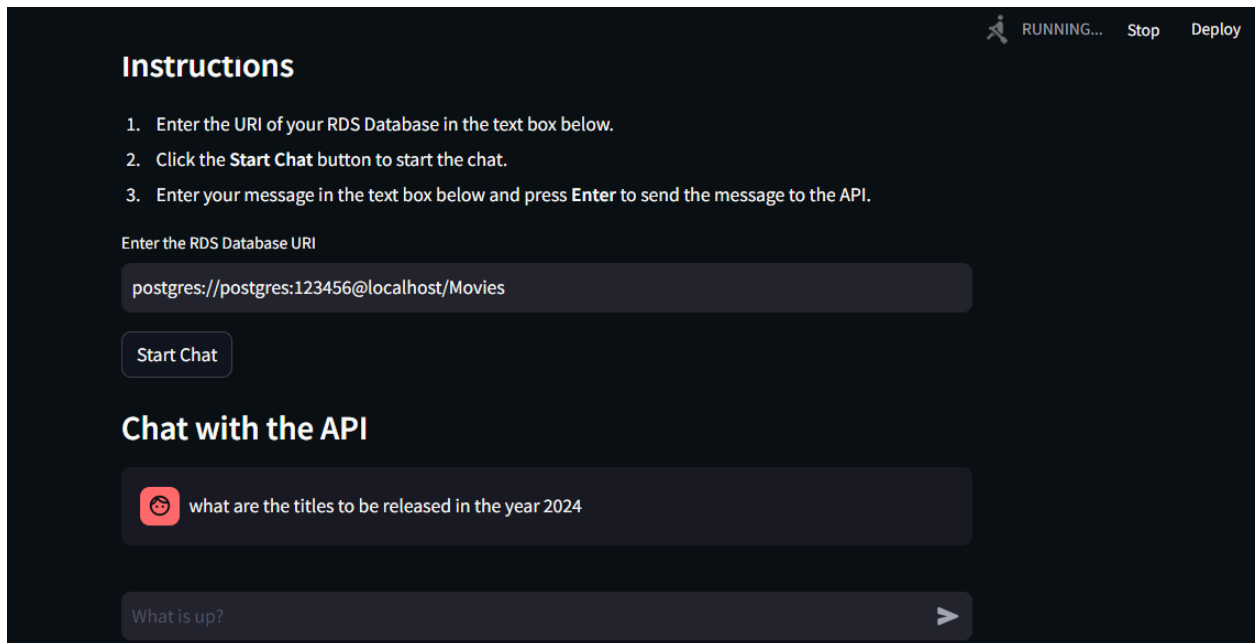
ix. foreign_key_info: This attribute is a string variable that holds information about foreign key constraints present in the database schema. It includes details such as the table containing the foreign key, the referenced table, and the columns involved in the foreign key relationship.

x. unique_id: This attribute is a string variable generated to provide a unique identifier for different database sessions or operations within the application. It's typically used for creating unique filenames, directories, or other identifiers to avoid naming conflicts. It helps in preventing conflicts and aiding in organization and management of database-related resources.

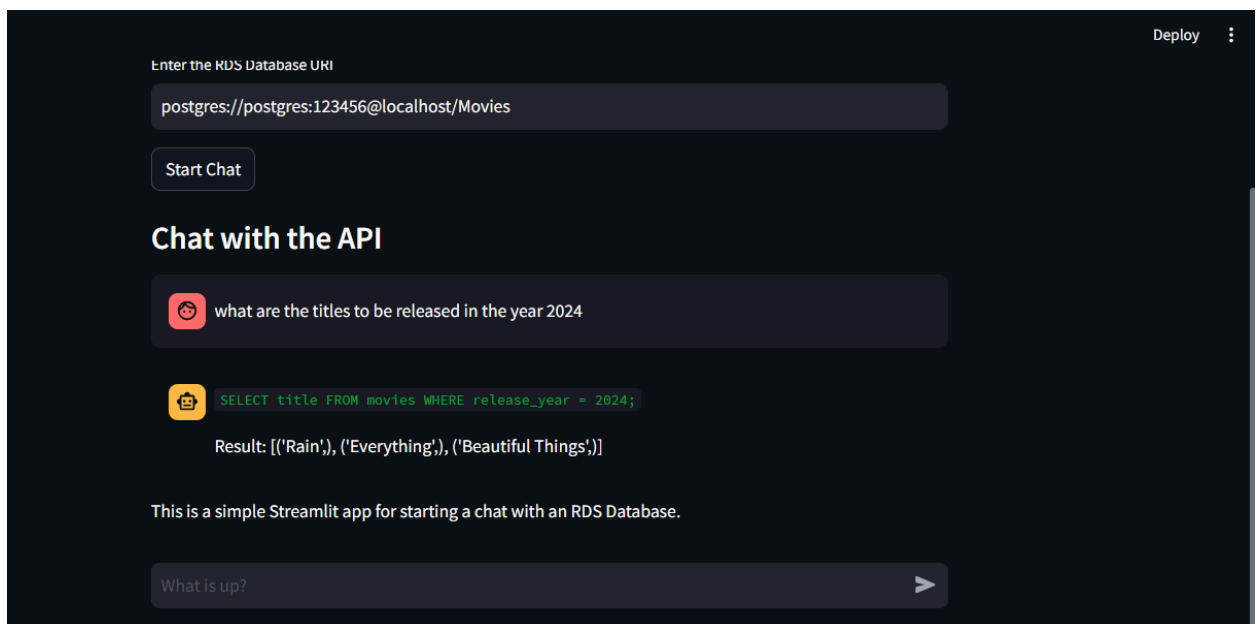
xi. table_info: This attribute is a string variable that holds detailed information about database tables, including table names, column names, and data types. The string is constructed by iterating over the tables retrieved from the database and appending relevant details. It's intended to provide users with a detailed description of the database schema, aiding in understanding the structure of the tables.

xii. vectordb: This attribute is utilized to store vector embeddings associated with database tables. These embeddings can be used for similarity search and retrieval operations. It enables the storage and retrieval of vector embeddings associated with database tables, facilitating similarity-based search operations.

5.3. Experimental Screenshots



Screenshot 5.1. Database URI and Question



Screenshot 5.2. Output

6. Experimental Setup

Used **OpenAI API Key**, **Jupyter Notebook**, **streamlit** to develop SQL generator. API key will be used to access LLM and fine-tune LLM, Python IDLE will be used to create the SQL Generator, and streamlit will be used to create interface for end user.

6.1. Obtain OpenAI API Key

To obtain an OpenAI API key, you need to follow these steps:

1. **Create an OpenAI Account:** If you haven't already, sign up for an account on the OpenAI website.
2. **Navigate to API Settings:** Log in to your OpenAI account and go to the API settings page. This is where you'll find your API key and manage your API usage.
3. **Generate an API Key:** If you haven't generated an API key yet, you'll need to create one. Click on the button or link to generate a new API key.
4. **Copy the API Key:** Once the API key is generated, you'll see it displayed on the screen. Copy this key to your clipboard.
5. **Store the API Key Securely:** It's essential to handle API keys securely. Avoid hardcoding API keys directly into your code, especially if you're sharing your code publicly. Instead, you can use environment variables or configuration files to store and access the API key.

6.2. Setup Python IDLE:

To install and set up Python IDLE, you can follow these steps:

1. **Install Python:** First download and install Python from the official Python website: python.org. Make sure to download the version appropriate for your operating system (Windows, macOS, or Linux). During installation, ensure that the option to install IDLE is selected.
2. **Launch IDLE:** Once Python is installed, you can launch IDLE. The method for launching IDLE varies depending on your operating system:

- a. **Windows:** Open the Start menu, type "IDLE", and select "IDLE (Python x.x)" from the search results.
 - b. **macOS:** Open a terminal window and type `idle` and press Enter.
 - c. **Linux:** Open a terminal window and type `idle` or `python3 -m idle` and press Enter.
3. **Using IDLE:** Once IDLE is launched, you'll see the Python shell window. You can start writing and executing Python code directly in the shell. Additionally, you can create and edit Python scripts by selecting "File" > "New File" from the menu.
4. **Customize Settings (Optional):** IDLE comes with a few customizable settings. You can adjust the font size, theme, and other preferences by going to "Options" > "Configure IDLE".
5. **Create a New File:** If you want to write a longer Python program, it's best to create a new file. You can do this by selecting "File" > "New File" from the menu.
6. **Write Your Python Code:** In the new file window, you can start writing your Python code.
7. **Save Your File:** Once you've written your code, you should save the file. Select "File" > "Save As..." from the menu, choose a location, and give your file a name with a ".py" extension. For example, you could name it "hello.py".
8. **Run Your Code:** After saving your file, you can run it by selecting "Run" > "Run Module" from the menu, or by pressing the F5 key. Alternatively, you can also click on the "Run" button in the toolbar.
9. **View Output:** After running your code, if your code requires user input, you can interact with it directly in the Python shell window and the output will be displayed in the Python shell window

6.3. Setup Streamlit

To install and set up Streamlit, a popular Python library for creating interactive web applications, follow these steps:

1. ***Install Python:*** Ensure you have Python installed on your system. You can download and install Python from the official Python website: <https://www.python.org/>. Make sure to check the option to add Python to your system PATH during installation.
2. ***Install Streamlit:*** You can install Streamlit using pip, the Python package manager. Open a terminal or command prompt and run the following command:
pip install streamlit.
3. ***Create a Streamlit Application:*** Once Streamlit is installed, you can create a new Python script (`.py` file) to define your Streamlit application. For example, create a file named `app.py`.
4. ***Write Streamlit Code:*** Write your Streamlit application code in `app.py`. Streamlit provides a simple and intuitive API for creating web applications directly from Python scripts.
5. ***Run Streamlit Application:*** In your terminal or command prompt, navigate to the directory containing `app.py` and run the following command: streamlit run app.py.
6. ***Access Your Streamlit App:*** Once the Streamlit server starts, it will provide a local URL (usually <http://localhost:8501>) where you can access your Streamlit application in your web browser.
7. ***Develop Your Application:*** You can continue to develop your Streamlit application by modifying `app.py`. Streamlit provides numerous components and features for building interactive data-driven applications, including widgets, charts, layouts, and more. Refer to the Streamlit documentation for detailed information: <https://docs.streamlit.io/>.
8. ***Deploy Your Application:*** Once you're satisfied with your Streamlit application, you can deploy it to various platforms, including Streamlit Sharing, Heroku, AWS, or any other hosting provider.

6.4. Libraries Used

6.4.1. OpenAI

OpenAI Installed enables access to OpenAI's large language models, such as GPT (Generative Pre-trained Transformer), for natural language processing tasks. This involves installing the necessary libraries and dependencies, obtaining an API key for authentication, integrating OpenAI's functionality into applications or workflows, and utilizing the models for tasks like text generation, summarization, translation, and more. Optional customization allows for fine-tuning or adapting models to specific requirements, empowering developers and researchers with state-of-the-art language processing capabilities.

6.4.2. LangChain

LangChain, an indispensable tool, provides functionalities for understanding and representing database schemas in a structured manner. This includes extracting information about tables, columns, data types, and relationships, which can be utilized for tasks like generating SQL queries or providing schema-related information to users. The provides utilities for generating vector embeddings of text data. Vectorization techniques are essential for representing textual information. It includes functionalities for persisting and managing data related to database schemas, queries, embeddings, and other artifacts. It enables using NLP techniques to enhance database interactions, it might offer features to generate SQL queries based on natural language prompts, understand user intents related to database operations, and retrieve relevant information from databases using natural language queries. In essence, LangChain emerges as an essential component for organizations seeking to streamline management workflows with precision and effectiveness.

6.4.3. Psycopg2

Psycopg2 serves as a bridge between Python applications and PostgreSQL databases, allowing developers to perform various database operations directly from their Python code. It facilitates establishing connections, executing SQL

queries, managing transactions, and fetching results from PostgreSQL databases within Python programs. Developers can create connections using the `'psycopg2.connect()'` function, passing connection parameters such as database name, user credentials, host, and port. After establishing a connection, it allows developers to create a cursor object. Cursors are used to execute SQL queries and manage the result sets returned by those queries. Psycopg2 allows fetching query results as Python objects, such as tuples or dictionaries, facilitating easy manipulation and processing of retrieved data.

6.4.4. Universally Unique Identifier (UUID)

UUID is a python module used for generating unique identifiers that are highly unlikely to collide with other identifiers generated in the same or different systems. Once generated, UUIDs are immutable and cannot be modified. This ensures their uniqueness and consistency across systems. UUIDs are represented in different formats, such as hexadecimal, binary, and string representations. They can also be formatted with or without hyphens for readability. It is used to generate unique identifiers (`unique_id`) for different database sessions or operations. These identifiers are used for creating unique filenames, directories, or other identifiers to avoid naming conflicts.

6.4.5. Pandas

Pandas is a powerful library for data manipulation and analysis in Python. It provides data structures like `DataFrame` for tabular data and functions for data cleaning, transformation, and analysis. The core data structure in pandas is the `DataFrame` which is used for loading, manipulating, and storing tabular data retrieved from the database. It's particularly useful for working with structured data such as table details and foreign key information. It provides a wide range of functions and methods for data manipulation, including indexing, slicing, filtering, merging, joining, grouping, aggregating, and reshaping data. Pandas supports reading and writing data in various formats, including CSV, Excel, SQL databases, JSON and HTML.

6.4.6. Chroma

The Chroma library is designed for vectorization and indexing of text-based data, with the goal of facilitating efficient retrieval and similarity search operations. It provides functionalities for converting textual data into numerical representations (vectors) using techniques such as word embeddings or other natural language processing (NLP) methods. Chroma is utilized for creating vector representations of table details obtained from a PostgreSQL database. These vector representations are likely used for indexing and searching relevant tables based on the similarity of their descriptions to user queries. It includes functions or classes for converting text data into vector representations. This process is essential for performing similarity-based searches and analyses. It also provides functionalities for building and managing vector indexes, enabling efficient retrieval of similar items based on the vector proximity or similarity measures. The library supports persistent storage of vector data and indexes, allowing them to be saved to disk and loaded efficiently for future use.

6.4.7. CSVLoader

The CSVLoader module class is designed to facilitate the loading of data from CSV (Comma-Separated Values) files into Python data structures. It provides utilities for converting CSV data into appropriate Python data structures, such as lists, dictionaries, or pandas DataFrames, depending on the requirements of the application. CSVLoader module is used for loading data from CSV files containing information about database tables and foreign keys. It is employed to read CSV files generated from database metadata and convert the tabular data into a format suitable for further processing, such as creating vector representations or generating SQL queries. It contains functions like reading the CSV files, parsing the data contained within them, and converting them into a suitable format for further processing or analysis. It also includes functionalities like handling different delimiters, and processing data rows and columns.

```

team17.py - C:\Users\Akshara\Desktop\team17.py (3.11.5)
File Edit Format Run Options Window Help

def get_the_output_from_llm(query, unique_id, db_uri):
    ## Load the tables csv
    filename_t = 'csvs/tables_' + unique_id + '.csv'
    df = pd.read_csv(filename_t)

    ## For each relevant table create a string that list down all the columns and their data types
    table_info = ''
    for table in df['table_name']:
        table_info += 'Information about table' + table + ':\n'
        table_info += df[df['table_name'] == table].to_string(index=False) + '\n\n\n'

    answer_to_question_general_schema = check_if_users_query_want_general_schema_information_or_sql(query)
    if answer_to_question_general_schema == "yes":
        return prompt_when_user_want_general_db_information(query, db_uri)
    else:
        vectordb = Chroma(embedding_function=embeddings, persist_directory="./vectors/tables_" + unique_id)
        retriever = vectordb.as_retriever()
        docs = retriever.get_relevant_documents(query)
        print(docs)

        relevant_tables = []
        relevant_tables_and_columns = []

        for doc in docs:
            table_name, column_name, data_type = doc.page_content.split("\n")
            table_name = table_name.split(":")[1].strip()
            relevant_tables.append(table_name)
            column_name = column_name.split(":")[1].strip()
            data_type = data_type.split(":")[1].strip()
            relevant_tables_and_columns.append((table_name, column_name, data_type))

    ## Load the tables csv
    filename_t = 'csvs/tables_' + unique_id + '.csv'
    df = pd.read_csv(filename_t)

    ## For each relevant table create a string that list down all the columns and their data types
    table_info = ''
    for table in relevant_tables:
        table_info += 'Information about table' + table + ':\n'
        table_info += df[df['table_name'] == table].to_string(index=False) + '\n\n\n'

```

Screenshot. 6.1. Coding environment screenshot

Instructions


1. Enter the URI of your RDS Database in the text box below.
2. Click the **Start Chat** button to start the chat.
3. Enter your message in the text box below and press **Enter** to send the message to the API.


Enter the RDS Database URI

postgres://postgres:123456@localhost/Movies

Start Chat

Chat with the API

 what are the titles to be released in the year 2024

 SELECT title FROM movies WHERE release_year = 2024;

Result: [('Rain'), ('Everything'), ('Beautiful Things')]

What is up?

Screenshot. 6.2. User Interface

The screenshot shows the pgAdmin 4 interface. On the left, the 'Object Explorer' pane displays the database structure, with 'iss_rec' table under the 'public' schema selected. The main pane shows a SQL query: `SELECT * FROM public.iss_rec ORDER BY iss_no ASC`. Below the query, the 'Data Output' tab displays the results of the query in a table format.

	iss_no [PK] integer	iss_date date	mem_no integer	book_no integer
1	900	2022-03-18	109	204
2	901	2022-03-19	102	209
3	902	2022-03-19	104	205
4	903	2022-03-20	100	200
5	904	2022-03-20	105	208
6	905	2022-03-21	101	201
7	906	2022-03-22	106	202

Screenshot. 6.3. Database

6.5. Parameters

The definition of precision score, Execution accuracy, and Semantic accuracy parameters is elucidated as follows.

Precision score refers to the ratio of correctly generated SQL queries to the total number of generated queries. It measures the accuracy of the generated queries. A high precision score indicates that the generator is generating accurate SQL commands in response to natural language queries, minimizing the chances of inaccurate queries, as illustrated in Equation (1).

$$\text{Precision Score} = \frac{\text{Number of correctly Generated Queries}}{\text{Total Number of Queries}} \quad (1)$$

Execution accuracy evaluates the generated SQL queries for their correctness in terms of executing against the database. Incorrect queries may lead to errors or undesired outcomes. This is crucial as it assesses whether the generated queries are syntactically

and semantically correct and whether they retrieve the desired information from the database. The equation for execution accuracy is presented in Equation (2).

$$\text{Execution Accuracy} = \frac{\text{No. of correctly executed queries}}{\text{total no. of queries}} \times 100 \quad (2)$$

Semantic accuracy assesses the extent to which the generated SQL queries convey the correct meaning or intent of the input natural language queries. Semantic accuracy helps ensure that the generated queries are not only grammatically correct but also functionally correct in terms of their intended purpose, it is demonstrated in Equation (3).

$$\text{Semantic Accuracy} = \frac{\text{Number of Queries With Correct Semantic Capture}}{\text{Total Number of Queries}} \times 100 \quad (3)$$

7. Results and Discussion

In order to evaluate the efficacy of system-generated queries, we conducted experiments with five distinct models, GNN [16], BERT [19], Semantic Parser [27], Seq2Seq [2], Rule-Based Generation [24] and our model- LLSG (Large Language model based SQL Generation) to assess their performance. Our evaluation entails the presentation of results Precision Score, Execution Accuracy and Semantic Accuracy.

1. Precision Score

It refers to the ratio of correctly generated SQL queries to the total number of generated queries. It measures the accuracy of the generated queries. A high precision score indicates that the generator is generating accurate SQL commands in response to natural language queries, minimizing the chances of inaccurate queries.

$$\text{Precision Score} = \frac{\text{Number of correctly Generated Queries}}{\text{Total Number of Queries}} \quad (1)$$

Table 6.1. depicts the average precision scores of various SQL query generation techniques applied to 50 different SQL queries. Our findings show that LLSG performs better than other approaches, with a precision score of 0.894.

Table 6.1. Average Precision Score of Various Models

S.no	Models	Average Precision Score
1	GNN	0.482
2	BERT	0.393
3	Semantic Parser	0.710
4	Seq2Seq	0.241
5	Rule Based Generator	0.358
6	LLSG	0.894

The precision score percentages for all six models are illustrated in Fig. 1.

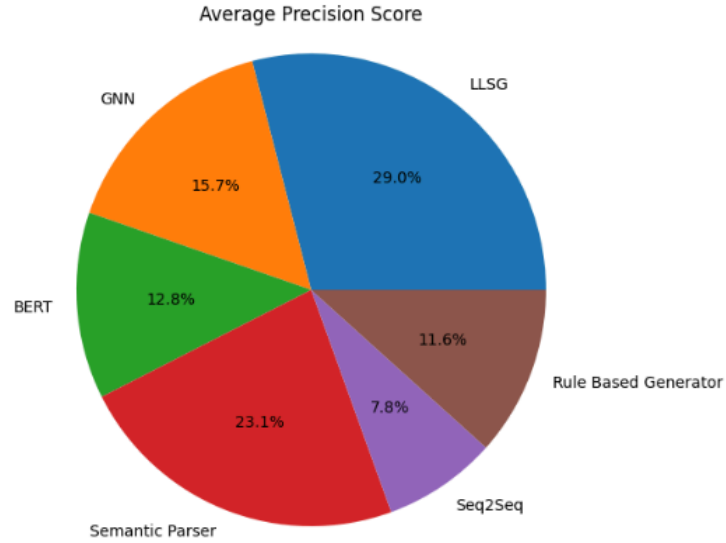


Figure 6.1. Precision Score of Models

2. Execution Accuracy

It evaluates the generated SQL queries for their correctness in terms of executing against the database. Incorrect queries may lead to errors or undesired outcomes. This is crucial as it assesses whether the generated queries are syntactically and semantically correct and whether they retrieve the desired information from the database.

$$Execution\ Accuracy = \frac{No.\ of\ correctly\ executed\ queries}{total\ no.\ of\ queries} \times 100 \quad (2)$$

Table 6.2. displays the Execution Accuracy for various models applied to a given set of generated queries. Examining the table, it is evident that the LLSG exhibits the highest Execution Accuracy.

Table 6.2. Average Execution Accuracy of Different Models

S.no	Models	Average Execution Accuracy
1	GNN	0.71
2	BERT	0.50
3	Semantic Parser	0.39
4	Seq2Seq	0.68
5	Rule-Based Generator	0.24
6	LLSG	0.92

The Execution Accuracy percentages for all six models are illustrated in Fig. 2.

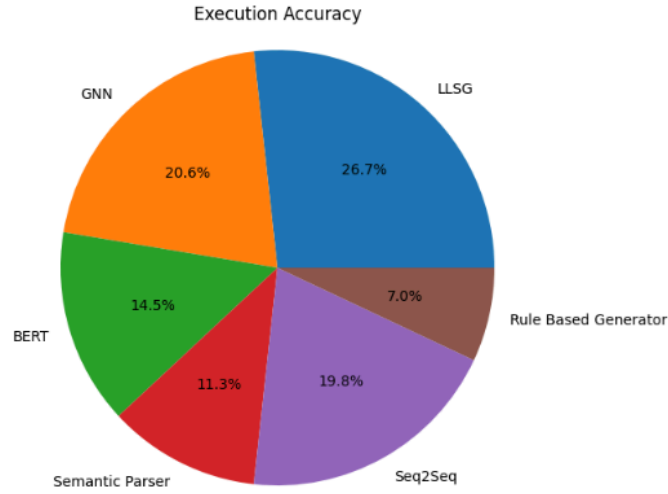


Figure 6.2. Execution Accuracy of Models

3. Semantic Accuracy

It assesses the extent to which the generated SQL queries convey the correct meaning or intent of the input natural language queries. Semantic accuracy helps ensure that the generated queries are not only grammatically correct but also functionally correct in terms of their intended purpose.

$$\text{Semantic Accuracy} = \frac{\text{Number of Queries With Correct Semantic Capture}}{\text{Total Number of Queries}} \times 100 \quad (3)$$

Table 6.3. displays the Semantic Accuracy for various models applied to a given set of generated queries. Examining the table, it is evident that the LLSG exhibits the highest Semantic Accuracy.

Table 6.3. Average Semantic Accuracy of Various Models

S.no	Models	Average Semantic Accuracy
1	GNN	0.41
2	BERT	0.60
3	Semantic Parser	0.84
4	Seq2Seq	0.32
5	Rule-Based Generator	0.29
6	LLSG	0.90

The Execution Accuracy percentages for all six models are illustrated in Fig. 3.

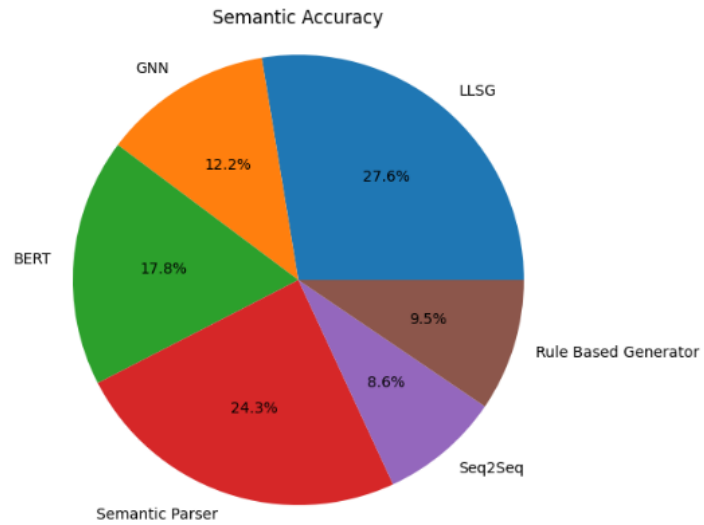


Figure 6.3. Semantic Accuracy of models

Upon thorough examination of the obtained outputs and results, it is evident that the LLGS provides concise and clear queries while capturing all the semantics, regardless of the dataset or table size.

8. Conclusion

The research findings highlight the efficacy of Large language model based SQL Generation (LLGS) in accurately distilling the essence of entire SQL queries, a task where other models often fall short. Unlike its counterparts, which struggle to produce concise queries and capture the semantics of the natural language, LLSG offers a solution that effectively surmounts these obstacles. By undergoing meticulous fine-tuning using a high-quality dataset, LLGS transcends execution constraints and can generate queries with varied datasets or table length. LLSG's strength lies in its ability to provide comprehensive and nuanced queries, thereby enhancing the generation process. By employing advanced techniques and harnessing the power of OpenAI's language model, LLSG captures the intricate details and essential elements of natural language with remarkable accuracy. Its capability to overcome dataset limitations ensures that it can encapsulate the overall gist of a language without sacrificing depth or clarity. The usage of the large language models and openAI enables LLSG to efficiently process large volumes of data, to understand the underlying semantics in the natural language. Additionally, the fine-tuning process enhances LLTS's understanding of ambiguity, syntax, and structure, enabling it to generate queries that faithfully represent the natural language question. Overall, LLSG represents a significant advancement in the field of SQL Generation, offering a solution that addresses the shortcomings of existing models and significantly improves the generation process. Its ability to generate comprehensive and nuanced queries underscores its effectiveness in facilitating efficient information retrieval from the database.

9. Future Enhancements

The LLSG model could encompass several avenues. Firstly, there's a potential for domain-specific fine-tuning, honing the model's understanding within specific natural language domains like semantic parsing or ambiguity. Additionally, extending LLSG to handle multiple database query generation could greatly benefit users dealing with databases. Implementing algorithms to optimize the generated SQL queries for efficiency, including query rewriting techniques and consideration of database indexes and statistics can help improve execution time and resource utilization. Develop robust error handling mechanisms to provide informative feedback to users when their queries cannot be translated accurately, suggesting revisions or alternatives to improve comprehension and generate correct SQL statements. Enabling seamless integration with popular data visualization tools like Tableau or Power BI, allows users to directly visualize the results of their SQL queries and explore insights from the data more interactively. Extending the system's capabilities to handle advanced SQL constructs and operations, including window functions, common table expressions (CTEs), nested queries, and complex joins, can accommodate more sophisticated analytical queries. Lastly, integrating LLSG with existing query research tools would streamline workflows for professionals and researchers without a technical background.

10. References

- [1] A. Deshpande, D. Kothari, A. Salvi, P. Mane and V. Kolhe, "Querylizer: An Interactive Platform for Database Design and Text to SQL Conversion," 2022 International Conference for Advancement in Technology (ICONAT), Goa, India, 2022, pp. 1-6, doi: 10.1109/ICONAT53423.2022.9725828.
- [2] T S, Anisha and P C, Rafeeque and Murali, Reena "Text to SQL Query Conversion Using Deep Learning: A Comparative Analysis", In proceedings of the International Conference on Systems, Energy & Environment (ICSEE) 2019, GCE Kannur, Kerala, July 2019
- [3] K. Lahoti, M. Paryani and A. Patil "Deep Learning Based Text to SQL Conversion on WikiSQL Dataset: Comparative Analysis", 2022 3rd International Conference on Issues and Challenges in Intelligent Computing Techniques , Ghaziabad, India, 2022, pp. 1-6, doi:10.1109/ICICT55121.2022.10064556.
- [4] A. Kate, S. Kamble, A. Bodkhe and M.Joshi "Conversion of Natural Language Query to SQL Query", 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 2018, pp. 488-491, doi: 10.1109/ICECA.2018.8474639.
- [5] M. Arefin, K.M. Hossen and M. N. Uddin "Natural Language Query to SQL Conversion Using Machine Learning Approach", 2021 3rd International Conference on Sustainable Technologies for Industry 4.0 (STI), Bangladesh, 2021, pp. 1-6, doi: 10.1109/STI53101.2021.9732586
- [6] P. Parikh et al., "Auto-Query - A simple natural language to SQL query generator for an e-learning platform," 2022 IEEE Global Engineering Education Conference (EDUCON), Tunis, Tunisia, 2022, pp. 936-940, doi: 10.1109/EDUCON52537.2022.9766617.
- [7] Xu, Xiaojun & Liu, Chang & Song, Dawn. (2017). "SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. "
- [8] T. Shi & Tatwawadi, Kedar & Chakrabarti, Kaushik & Mao, Yi & Polozov, Oleksandr & Chen, Weizhu. (2018),"IncSQL: Training Incremental Text-to-SQL Parsers with Non-Deterministic Oracles".

- [9] C.Wang, Kedar Tatwawadi, Marc Brockschmidt, Po-Sen Huang, Yi Mao, Oleksandr Polozov and Rishabh Singh. “Robust Text-to-SQL Generation with Execution-Guided Decoding.”
- [10] Moses Visperas, Aunhel John Adoptante, Christalline Joie Borjal, Ma. Teresita Abia, Jasper Kyle Catapang, Elmer Peramo, “On Modern Text-to-SQL Semantic Parsing Methodologies for Natural Language Interface to Databases: A Comparative Study”, 2023 International Conference on Artificial Intelligence in Information and Communication (ICAIIIC), pp.390-396, 2023.
- [11] C.Wang& Cheung, Alvin & Bodik, Rastislav. (2017), “Synthesizing highly expressive SQL queries from input-output examples”, pp.452-466, doi: 10.1145/3062341.3062365.
- [12] Marc Brockschmidt, C.Wang and Rishabh Singh. “Pointing Out SQL Queries From Text”, (2018).
- [13] N.Yaghmazadeh, Navid & Wang, Yuepeng & Dillig, Isil & Dillig, Thomas. (2017), “SQLizer: query synthesis from natural language”, vol. 1, pp.1-26, doi: 10.1145/3133887.
- [14] Po-Sen Huang, Chenglong Wang, Rishabh Singh and Wen-tau Yih "Natural Language to Structured Query Generation via Meta-Learning", 2018.
- [15] J. M. Zelle and R. J. Mooney, “Learning to parse database queries using inductive logic programming”, in Proc. 13th Nat. Conf. Artif. Intell., vol. 2, 1996, pp. 1050–1055.
- [16] B. Bogin, M. Gardner, and J. Berant, “Representing schema structure with graph neural networks for Text-to-SQL parsing” in Proc. 57th Annual Meeting Assoc. Computer. Linguistics 1, 2019, pp.4560–4565.
- [17] C.Sugandhika and S.Ahangama, "Heuristics-Based SQL Query Generation Engine," 2021 6th International Conference on Information Technology Research (ICITR), Moratuwa, Sri Lanka, 2021, pp. 1-7, doi: 10.1109/ICITR54349.2021.9657317.
- [18] A. Anisyah, T. E. Widagdo and F. Nur Azizah, "Natural Language Interface to Database (NLIDB) for Decision Support Queries", 2019 International

Conference on Data and Software Engineering (ICoDSE), Pontianak, Indonesia, 2019, pp. 1-6, doi: 10.1109/ICoDSE48700.2019.9092769.

[19] X. Zhang, F. Yin, G. Ma, B. Ge and W. Xiao, "M-SQL: Multi-Task Representation Learning for Single-Table Text2sql Generation," in IEEE Access, vol. 8, pp. 43156-43167, 2020, doi: 10.1109/ACCESS.2020.2977613

[20] I.AndroutsopoulosI, G.Ritchie & P.Thanisch "Masque/sql-An Efficient and Portable Natural Language Query Interface for Relational Databases".

[21] Muhammad Shahzaib Baiget, Azhar Imran, Aman Ullah Yasin, Abdul Haleem Butt & Muhammad Imran Khan "Natural Language to SQL Queries: A Review", International Journal of Innovations in Science and Technology, 2022, vol. 4, pp.147-162, doi:10.33411/IJIST/2022040111.

[22] B.Sujatha, S.Vishwanadha Raju "Natural Language Query Processing for Relational Database using EFFCN Algorithm",International Journal of Computer Sciences and Engineering, 2016, vol.4.

[23] Abhishek Kharade "Generation of SQL Query Using Natural Language Processing",2023.

[24] Adrián Bazaga, Nupur Gunwant & Gos Micklem "Translating synthetic natural language to database queries with a polyglot deep learning framework", 2021, vol.11, pp.18462, doi:10.1038/s41598-021-98019-3.

[25] Niculae Stratica, Leila Kosseim and Bipin C. Desai "NLIDB Templates for Semantic Parsing", 8th International Conference on Applications of Natural Language to Information Systems, Germany, 2003, pp.235-241

[26] T.H.Y. Vuong, T.T.T. Nguyen, N.T. Tran, L.M. Nguyen and X.H.Phan, "Learning to Transform Vietnamese Natural Language Queries into SQL Commands," 2019 11th International Conference on Knowledge and Systems Engineering (KSE), Vietnam, 2019, pp. 1-5, doi: 10.1109/KSE.2019.8919393.

[27] Gauri Rao, Chanchal Agarwal, Snehal Chaudhry, Nikita Kulkarni and S.H. Patil "Natural Language Query Processing using Semantic Grammar", International Journal on Computer Science and Engineering, 2010, vol.2, pp. 219-223.

- [28] P. Pasupat and P. Liang, "Compositional semantic parsing on semi-Structured tables", 2015, pp. 1470–1480, doi:10.3115/v1/P15-1142.
- [29] Amit Pagrut, Shambhoo Kariya, Vibhavari Kamble and Ishant Pakmode," Automated SQL Query Generator by Understanding a Natural Language", International Journal on Natural Language Computing, vol.7, pp.01-11, doi:10.5121/ijnlc.2018.7301.
- [30] R. Kumar and M. Dua, "Translating controlled natural language query into SQL query using pattern matching technique," International Conference for Convergence for Technology-2014, Pune, India, 2014, pp. 1-5, doi: 10.1109/I2CT.2014.7092161.
- [31] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating Structured Queries From Natural Language Using Reinforcement Learning," in 2018 International Conference on Learning Representations (ICLR), Vancouver, BC, Canada, 2018
- [32] T. Yu, D. Wang, Matthew Purver, Z. Li and J. Ma et. al, "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task," in Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2018
- [33] J. Guo, Z. Zhan, Y. Gao, Y. Xiao, J.-G. Lou, T. Liu, and D. Zhang, "Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation"
- [34] H. Long and D. Cao, "Bert-based Text-to-SQL Generation method with question-table content enhancement and template filling," 2021 2nd International Conference on Information Science and Education (ICISE-IE), Chongqing, China, 2021, pp. 969-973, doi: 10.1109/ICISE-IE53922.2021.00222
- [35] S. S. Badhya, A. Prasad, S. Rohan, Y. S. Yashwanth, N. Deepamala and G. Shobha, "Natural Language to Structured Query Language using Elasticsearch for descriptive columns," 2019 4th International Conference on Computational Systems and Information Technology for Sustainable Solution (CSITSS), Bengaluru, India, 2019, pp. 1-5, doi: 10.1109/CSITSS47250.2019.9031030.
- [36] Richard Socher and Dragomir Radev, "Editing-Based SQL Query Generation for Cross-Domain Context-Dependent Questions"

- [37] Oleksandr Polozov, Matthew Richardson, B. Wang, R. Shin, X. Liu, "RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers," in Proceedings of the AAAI Conference on Artificial Intelligence (AAAI), vol. 35, no. 18, pp. 1673-1681, May 2021, doi: 10.1609/aaai.v35i18.16529.
- [38] Matthew Purver, John R. Woodward, John Drake, Yujian Gan, Xinyun Chen, Jinxia Xie, "Natural SQL: Making SQL Easier to Infer from Natural Language Specifications"
- [39] R. Zhong, T. Yu, and D. Klein, "Semantic Evaluation for Text-to-SQL with Distilled Test Suites," in Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: Student Research Workshop, Florence, Italy, 2019, pp. 154-159
- [40] G. Swain, Object-Oriented Analysis and Design Through Unified Modelling Language. Laxmi Publication, Ltd., 2010.
- [41] G. Booch, D. L. Bryan, and C. G. Peterson, Software engineering with Ada. Redwood city, Calif.: Benjamin/Cummings, 1994.