CHAPTER 1

INTRODUCTION

1.1 PROJECT OVERVIEW

In today's data-driven world, structured databases play a vital role in storing, organizing, and managing information across a wide range of domains. However, interacting with these databases typically requires knowledge of Structured Query Language SQL, which poses a challenge for users who lack technical expertise. The project titled "TEXT-TO-SQL: AI-Based Query Conversion System" addresses this issue by providing an intelligent solution that allows users to query relational databases using natural language. This system leverages the capabilities of Natural Language Processing NLP and advanced large language models to automatically convert human language queries into executable SQL statements.

The project utilizes Google’s Gemini API to interpret natural language queries and translate them into syntactically correct SQL, which is then executed on a backend SQLite database. The output of each query is presented to the user through a web-based Graphical User Interface GUI built with Streamlit. This interface supports both real-time query input and batch processing of queries, offering an accessible, user-friendly platform that does not require prior programming knowledge. By combining natural language understanding with seamless SQL execution, the system ensures that non-technical users can gain insights from structured datasets with ease and efficiency.

1.2 UNDERSTANDING TEXT-TO-SQL SYSTEMS

Text-to-SQL systems are designed to enable intuitive access to database systems by translating plain English queries into formal SQL commands. These systems rely on several key processes, including natural language understanding, semantic parsing, and schema awareness. The goal is to bridge the gap between user intent and structured database retrieval by accurately identifying the entities, attributes, and relationships described in natural language.

With the rise of large-scale pre-trained language models, Text-to-SQL systems have significantly improved in terms of precision, contextual understanding, and generalization across various database schemas. Modern models are capable of understanding a wide variety of query formulations and mapping them to the appropriate SQL structure. As a result, they enable more natural interactions between users and databases, promoting wider adoption and more efficient data utilization.

1.3 CHALLENGES IN NATURAL LANGUAGE TO SQL CONVERSION

Translating natural language into SQL is a complex and multi-faceted task. One major challenge is the inherent ambiguity of human language, where the same query can be expressed in many different ways. This requires the system to be capable of understanding a wide range of linguistic variations and mapping them to consistent SQL structures. Another challenge is the dependency on database schema, where the model must be aware of table names, column names, and relationships to produce syntactically and semantically valid queries.

In addition, complex queries involving nested clauses, aggregations, joins, and subqueries demand advanced reasoning and semantic parsing capabilities.

The system must also be resilient to grammatical errors and informal language usage, which are common in real-world input. Handling incorrect or unsupported queries gracefully and providing meaningful feedback to the user is another critical requirement. This project addresses these challenges through the use of the Gemini API, which offers state-of-the-art language understanding and contextual reasoning to generate accurate SQL queries from diverse natural language inputs.

1.4 NATURAL LANGUAGE PROCESSING IN TEXT-TO-SQL

Natural Language Processing plays a foundational role in the Text-to-SQL conversion process. It enables the system to interpret user inputs and extract relevant syntactic and semantic information needed for accurate SQL generation. The system first processes the input text by breaking it down into tokens, identifying key entities, and understanding the intent of the user. It then maps this understanding to the structure and vocabulary of the target database.

Advanced NLP models like Gemini use deep contextual embeddings to capture the meaning of each word within its sentence, considering both preceding and following words. This bidirectional understanding allows the model to handle linguistic subtleties, such as polysemy, idiomatic expressions, and indirect references. NLP also facilitates schema linking, where terms used in the user query are matched with relevant elements in the database schema. This process allows the model to produce queries that are not only syntactically correct but also semantically aligned with the user's intent and the database structure.

1.5 ROLE OF GEMINI API AND STREAMLIT UI

The Gemini API serves as the core of the query translation engine in this system. As a powerful language model, Gemini is capable of interpreting a wide range of natural language inputs and transforming them into accurate and executable SQL queries. Its ability to understand context, handle ambiguity, and generate structured output makes it highly suitable for this application. By fine-tuning the input prompts and providing schema guidance, the system ensures that the API produces results that are valid for the specific database in use.

The front-end interface is developed using Streamlit, a Python-based framework that allows rapid development of interactive web applications. This interface provides a simple and intuitive platform for users to input their natural language queries and view the corresponding SQL statements along with the query results. It also includes features such as query history, CSV upload for batch queries, and error handling to enhance the overall user experience. The integration of Gemini with Streamlit results in a seamless end-to-end system for AI-powered database querying.

1.6 OBJECTIVE OF THE PROJECT

The main objective of the "TEXT-TO-SQL: AI-Based Query Conversion System" is to democratize access to relational data by eliminating the need for manual SQL coding. By allowing users to interact with databases through natural language, the system reduces the technical barrier to data retrieval and analysis. It seeks to enhance productivity for users across various domains, including business, education, and research, where quick access to structured data is essential.

The project aims to provide a highly accurate and responsive Text-to-SQL pipeline that leverages state-of-the-art NLP technologies. It focuses on creating a system that can interpret complex queries, generate correct SQL commands, and return meaningful results, all within a user-friendly interface. In addition, the system is designed to be scalable and adaptable, paving the way for future enhancements such as multi-database support, advanced query optimization, and integration with voice-based input systems. Through these objectives, the project contributes to making intelligent database interaction more accessible, efficient, and impactful.

LITERATURE SURVEY

Several recent studies have explored the integration of Natural Language Processing and deep learning techniques to bridge the gap between natural language input and structured database querying. These contributions form the foundation upon which this project is built, and they offer insights into the evolution of Text-to-SQL systems and related challenges in the domain.

2.1 RAT-SQL: RELATION-AWARE SCHEMA ENCODING AND LINKING FOR TEXT-TO-SQL PARSERS

Author: Bailin Wang

Year: 2020

et al., published in, presents a significant advancement in neural semantic parsing for Text-to-SQL tasks. This work introduces RAT-SQL, a system that encodes both the database schema and the input query using a relation-aware self-attention mechanism. The model effectively links components of the natural language query to specific parts of the database schema, allowing for accurate generation of SQL statements. The authors demonstrate the model's strong performance on the Spider dataset, a complex benchmark involving cross-domain Text-to-SQL tasks. This research illustrates the importance of schema understanding and contextual linkage in improving translation accuracy and handling complex queries with multiple tables and nested operations.

2.2 EXECUTING SQL QUERIES DURING SEMANTIC PARSING IMPROVES ACCURACY

Author: Tobias Herzig

Year: 2021

Tobias Herzig et al., This research addresses the challenge of generating syntactically valid and executable SQL queries during the parsing process. The authors propose Picard, a system that integrates execution feedback into the training loop of Text-to-SQL models. By executing partial queries during decoding, Picard can reject invalid SQL paths early and guide the model toward generating more accurate and semantically correct outputs. This technique leads to significant improvements in exact match accuracy and execution accuracy on the Spider benchmark. The study emphasizes that combining semantic parsing with runtime validation enhances the robustness and reliability of Text-to-SQL systems, particularly in scenarios involving dynamic or ambiguous queries.

2.3 LEVERAGING PRETRAINED LANGUAGE MODELS FOR TEXT-TO-SQL TASKS

Author: Xiaojun Xu

Year: 2020

Xiaojun Xu et al., published in. This work investigates the application of pretrained transformer-based language models, such as BERT and RoBERTa, to semantic parsing tasks that require SQL generation. The authors introduce a system that integrates a pretrained language model into a sequence-to-sequence architecture with schema linking and pointer mechanisms. The system is trained and evaluated on multiple Text-to-SQL datasets, demonstrating that language models significantly improve query understanding and SQL generation quality.

Their experiments show that pretrained embeddings provide rich contextual representations that are crucial for handling linguistic variability in natural language inputs. This research supports the growing trend of using large-scale pretrained models to enhance performance across NLP-driven tasks, including database querying.

Together, these studies reflect the rapid evolution and innovation within the Text-to-SQL research domain. They highlight key areas of focus such as schema encoding, runtime execution, and the incorporation of language models. By building upon these advancements, the present project adopts a practical approach that integrates Google's Gemini API to leverage the power of large language models for natural language understanding, and uses SQLite as the backend for executing the generated SQL queries. Furthermore, the deployment through Streamlit ensures accessibility and interactivity, bringing theoretical innovations into a usable real-world system.

2.4 BRIDGE: A GRAPH-BASED SEMANTIC PARSER FOR MULTI-TABLE TEXT-TO-SQL TASKS

Author: Xi Victoria Lin
Year: 2020

BRIDGE introduces a novel graph-based approach to semantic parsing that handles complex SQL queries across multiple tables. The model uses a schema graph that connects tables and columns based on database relationships, enabling better encoding of relational structures. The authors also apply pre-trained BERT embeddings to encode both natural language queries and database schema components. BRIDGE performs well on the Spider dataset, demonstrating that graph-based encoding significantly boosts performance when dealing with multi-

SYSTEM ANALYSIS

3.1 EXISTING SYSTEMS

Existing systems that attempt to bridge natural language and database querying are often rigid and limited in capability. They typically rely on predefined templates or require users to frame their questions in specific formats. Most of these systems use basic rule-based natural language processing or simple keyword extraction methods, which do not scale well with complex or ambiguous queries. Additionally, many traditional tools fail to understand the true semantics behind a user's input, leading to inaccurate or incomplete SQL query generation. These systems also often lack integration with a live backend, making real-time data retrieval and dynamic interactions difficult to achieve.

3.2 DISADVANTAGES OF EXISTING SYSTEMS

The main drawback of existing systems is their inability to interpret flexible or natural human language effectively. They often demand structured inputs, which limits usability for non-technical users. Furthermore, the lack of transparency in the generated SQL query reduces user trust, as users cannot verify or understand the transformation process. These systems rarely include real-time query execution, meaning users must copy generated SQL into a separate database interface. Additionally, the process of uploading or managing new data usually requires manual SQL commands, making it inaccessible for users without technical skills.

3.3 PROPOSED SYSTEM

The proposed system, named "Text to SQL", provides a comprehensive solution that overcomes the limitations of traditional systems. It leverages the Gemini 2.0 Flash API to convert natural language queries into SQL statements, offering a more intelligent and flexible understanding of user input. The system is built with a Streamlit-based frontend, allowing users to interact through a clean and intuitive interface. When a user enters a question, the backend processes it via the Gemini API, converts it into a SQL query, and executes it on a MySQL database hosted locally. The results are then displayed in real time within the same interface. In addition to this, the system allows users to view the generated SQL query for transparency. It also includes an admin panel where users can upload CSV files to the database with a single click, eliminating the need to write SQL insert commands manually. Moreover, users can download the output data as a CSV file for external use.

3.4 ADVANTAGES OF PROPOSED SYSTEM

The Text to SQL system introduces a number of key improvements over existing tools. It allows users to enter natural, flexible questions without any need for programming knowledge. These inputs are processed and transformed into accurate SQL queries, which are then executed instantly to fetch results. Users can view the corresponding SQL statements, making the system more trustworthy and educational. The integrated admin panel simplifies data management by enabling CSV uploads directly to the database, thus supporting easy scalability. The ability to download query results as a CSV file further enhances usability. Overall, this system ensures an accessible, transparent, and efficient workflow from question input to result output.

SYSTEM SPECIFICATION

4.1 FUNCTIONAL REQUIREMENTS

- Converts natural language queries into valid SQL statements.
- Accepts user queries via a Streamlit-based web interface.
- Uses Gemini 2.0 Flash API to translate natural language into SQL.
- Executes the SQL query on a local MySQL server.
- Displays output results in the frontend interface.
- Shows the generated SQL query to the user.
- Allows users to download query results as a CSV file.
- Includes an admin panel for uploading new data via CSV.
- Provides example queries in a sidebar for user guidance.

4.2 NON-FUNCTIONAL REQUIREMENTS

- User-friendly and responsive interface using Streamlit.
- Fast response times with Gemini API and local MySQL.
- Modular architecture for easy upgrades and maintenance.
- Admin panel access restricted to authorized users.
- Handles incorrect or ambiguous queries gracefully.
- Designed to operate reliably under normal usage conditions.

4.3 HARDWARE AND SOFTWARE REQUIREMENTS

- **Hardware:**
    - Minimum Intel i5 processor or equivalent.
    - At least 8 GB of RAM.
    - Sufficient storage for CSV files and MySQL server.
    - Stable internet connection for Gemini API access.
- **Software:**
    - Python as the core programming language.
    - Streamlit for the frontend UI.
    - MySQL as the database system.
    - Gemini 2.0 Flash API for query understanding.
    - Supporting libraries: pandas, mysql-connector-python, csv.
    - Compatible with Windows, macOS, and Linux OS.

SYSTEM DESIGN

5.1 ARCHITECTURAL DESIGN

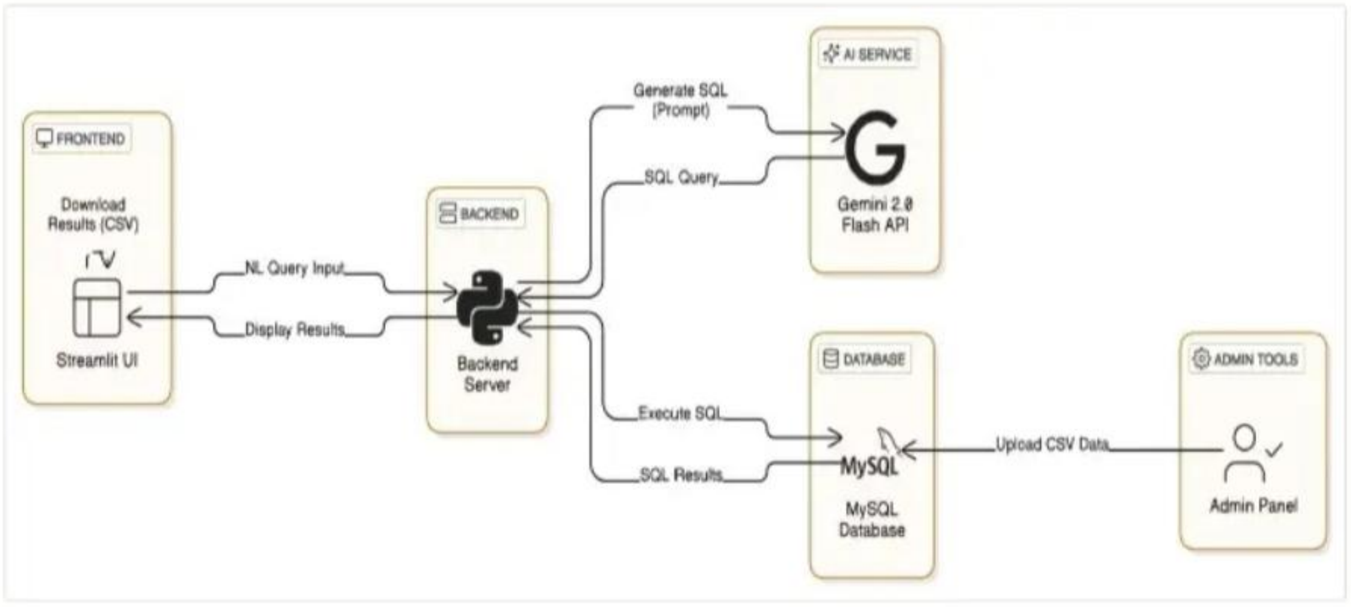

Figure 1: ARCHITECTURE DIAGRAM

5.2 DATA FLOW DESIGN:

The data flow of the system follows a sequential pipeline, ensuring smooth transition from natural language input to final SQL query execution and result display. The system's logic is organized to process the user input effectively and return the desired database output. The flow of data is described as follows:

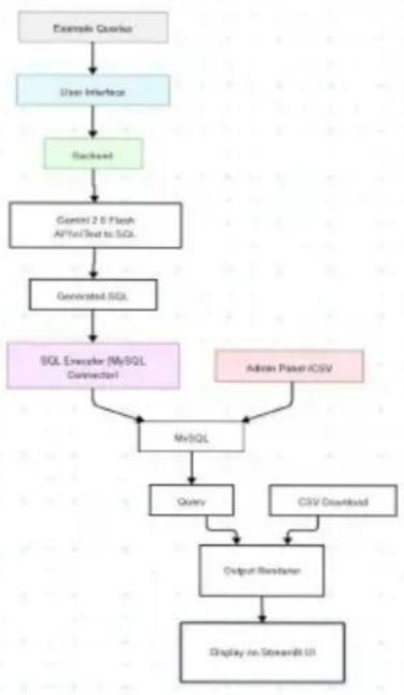

Figure 1: FLOW DIAGRAM

1. Input:

   The user enters a natural language question (e.g., "Show me all customers who ordered in March") through the Streamlit-based web interface.

2. Request Handling:

   The input is passed to the backend controller written in Python, which formats the prompt appropriately for the Gemini 2.0 Flash API.

3. Prompt Conversion (Gemini 2.0 Flash API):

   The formatted natural language query is sent to the Gemini API, which converts it into an equivalent SQL query string.

4. SQL Execution:

   The generated SQL query is validated and executed on the MySQL local server, where the database tables are stored.

5. Result Processing:

   The query output is retrieved from the MySQL server and prepared for display. It is converted into a structured table format.

6. Display Output:

   The result is shown on the Streamlit interface in a readable tabular format. The corresponding SQL query is also displayed for user reference.

7. CSV Export:

   Users are given the option to download the query results as a CSV file for further use.

8. Admin Upload Flow (Alternate Path):

   From the admin panel, a user can directly upload new datasets (in CSV format), which are parsed and added to the MySQL database without writing any SQL manually.

5.3 COMPONENT DESIGN

The proposed system is composed of several interdependent components, each responsible for a specific task in the natural language to SQL query generation workflow. At the forefront is the Streamlit-based frontend, which serves as the user interface. This component handles user inputs such as natural language queries and provides an intuitive, responsive layout for displaying the generated SQL query and the resulting output from the database. It also includes a sidebar with example queries to guide users in formulating their inputs effectively.

The input processing and Gemini integration module plays a crucial role in converting the user's natural language query into an SQL statement. This module receives the user input, formats it appropriately, and communicates with the Gemini 2.0 Flash API, which generates the corresponding SQL query using advanced natural language understanding and prompt engineering.

Once the SQL query is generated, the backend execution engine is responsible for forwarding it to the local MySQL database server, where it is executed against the current dataset. The backend manages the communication with the database, retrieves the results, and prepares the output for rendering.

The output handling component receives the query result and formats it into a tabular structure for display on the Streamlit interface. It also provides users with the option to download the results as a CSV file, enhancing usability and data portability. Additionally, the generated SQL query is shown alongside the result for transparency and educational purposes.

TESTING AND RESULTS

7.1 TESTING

7.1.1 Code Review

The codebase, including modules for user input handling, Gemini API integration, SQL execution, and result rendering, was manually reviewed. This review helped detect syntax errors, improper function calls, and logic flaws. Special attention was paid to API request formatting and database interaction commands to prevent runtime exceptions or invalid query errors.

7.1.2 Unit Testing

Each module was tested independently. The Query Generation Module was verified using mock inputs to ensure it produced correct SQL syntax. Similarly, the SQL Execution Module was tested with both valid and edge-case SQL queries to validate database connectivity and query handling. The CSV download function was tested to confirm that generated files reflected the correct query output.

7.2 Integration Testing

All system components were tested together as an end-to-end pipeline. From receiving the natural language input through the frontend to displaying and downloading the result, the complete process was simulated multiple times using varied queries. This ensured that each module worked seamlessly with others and maintained consistent data flow.

7.2.1 UI Testing

The Streamlit UI was tested on multiple browsers and screen sizes to verify responsiveness, correct rendering of query results, and smooth file downloads. The example queries on the sidebar were tested to confirm proper auto-filling and response behavior.

7.2.2 Error Handling Validation

Scenarios such as invalid natural language input, empty queries, malformed SQL, and MySQL connection errors were tested. Appropriate error messages and fallback mechanisms were confirmed to ensure user-friendly handling of exceptions.

7.3 RESULTS

7.3.1 STUDENT table data

NAME	CLASS	SECTION	MARKS
Alice Smith	10	A	89
Bob Johnson	10	A	76
Charlie Lee	10	B	92
Daisy Patel	11	A	85
Ethan Brown	11	B	70

7.3.2 ATTENDANCE table data

NAME	DATE	STATUS
Alice Smith	15-03-2024	Present
Bob Johnson	15-03-2024	Absent
Charlie Lee	15-03-2024	Present
Daisy Patel	15-03-2024	Absent
Ethan Brown	15-03-2024	Present

7.3.3 COURSE table data

CLASS	COURSE_NAME	INSTRUCTOR
10	Mathematics	Dr. Sharma
10	Science	Ms. Rao
11	English	Mr. Iyer
11	Physics	Dr. Verma

The system was evaluated based on its ability to translate natural language into accurate SQL queries and return correct data from the MySQL database. Multiple query types such as aggregation, filtering, sorting, and table joins were used for testing. The generated SQL queries closely matched expected outputs, and the results were displayed correctly on the Streamlit interface.

Key functionalities like SQL generation, CSV download, and admin-side CSV upload were all verified to work correctly. Response times for most queries were under two seconds, demonstrating the efficiency of Gemini 2.0 Flash API and the lightweight design of the Streamlit UI. Test cases involving invalid or ambiguous queries produced appropriate warnings, and in most cases, the system was able to recover or prompt the user to re-enter the input.