LLM SQL Query Generator Project Report

Sai Mounya Dumpala (22CS30023)

Contents

1 Problem Statement

Structured databases form the foundation of modern data-driven decision-making processes. These systems are highly efficient at storing, managing, and retrieving large volumes of data in an organized manner. However, the complexity of Structured Query Language (SQL) poses a significant challenge for non-technical users, such as analysts, managers, and students, who may lack programming experience or database knowledge.

The steep learning curve of SQL—especially in understanding syntax rules, filtering conditions, join operations, and schema navigation—creates a bottleneck for data accessibility. This often requires dependency on database administrators or software developers, leading to delays and reduced agility in decision-making.

This project aims to bridge this gap by building an intelligent interface that allows users to pose questions in natural language (e.g., English) and obtain both the translated SQL query and the corresponding output from the database. By integrating Google's Gemini Large Language Model (LLM), our system can interpret the user's intent, map it to relevant database operations, and return accurate results—all in real-time.

The tool democratizes access to structured data and eliminates the need to learn SQL, opening avenues for use in:

- Data Analytics: Quick and intuitive data retrieval for reports.
- Education: Aids students in understanding query formulation.
- Internal Tools: Enhances usability of dashboards in companies.
- Research: Facilitates easy exploration of tabular datasets.

2 Methodology

2.1 Environment Setup

To ensure the secure and maintainable operation of the system, environment variables are stored and managed using the `dotenv` library in Python. This modular setup helps in separating configuration from code and avoids hard-coding sensitive information.

Sensitive configuration includes:

- Gemini API Key
- PostgreSQL credentials (host, port, user, password, database name)

The environment variables are defined in a `.env` file which is loaded using:

```
from dotenv import load_dotenv
load_dotenv()
```

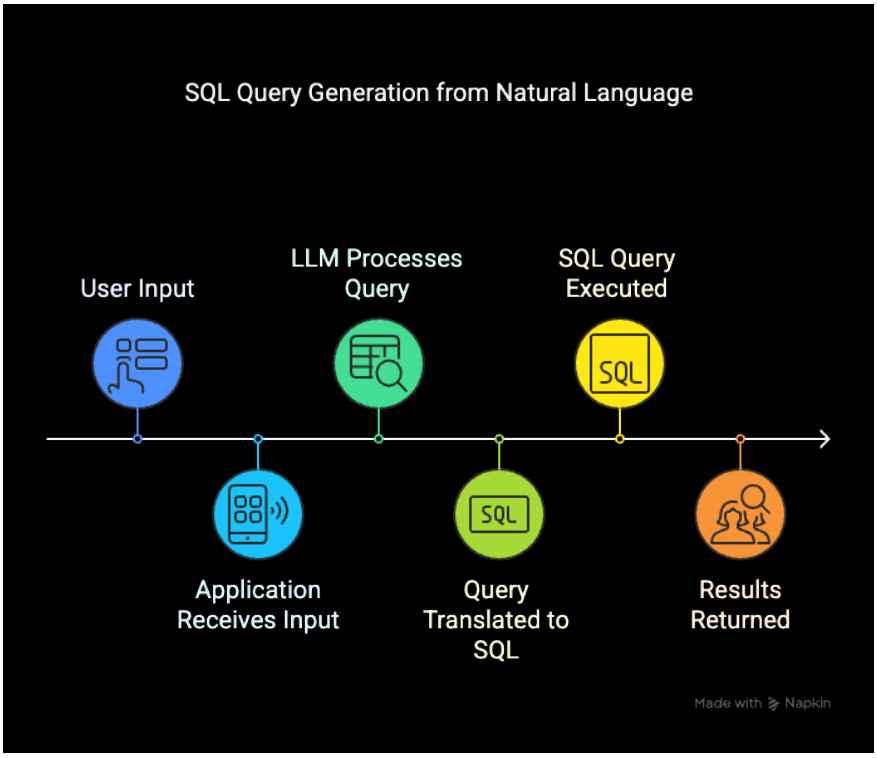
This approach improves portability and adheres to security best practices.

2.2 User Interface (Streamlit)

The system's frontend is built using Streamlit, an open-source Python framework for rapid web app development. Streamlit provides a clean, interactive interface with minimal overhead and supports real-time execution of Python scripts.

Key Features:

- **Text Box:** Users can input natural language queries.



- **Dropdown Menu:** Explore pre-defined example questions for quick testing.
- **Navigation Buttons:** Toggle between "Home" and "About" pages.
- **Accuracy level :** Users can select the accuracy level they want for the query generation. It has the following options :- Precise, Balanced and Creative
- **Show Explanation button :** Displays the logic used to convert the text to SQL query.
- **Result Display Area:** Render the generated SQL and the database results in tabular format.

Streamlit is ideal for prototyping due to its:

- Tight integration with Python backends
- Instant feedback loop
- Responsive layouts without writing JavaScript or HTML manually

2.3 LLM Integration using Google Gemini

To translate user queries into SQL, we employ Google's Gemini LLM.

- `gemini-2.0-flash` (for speed)

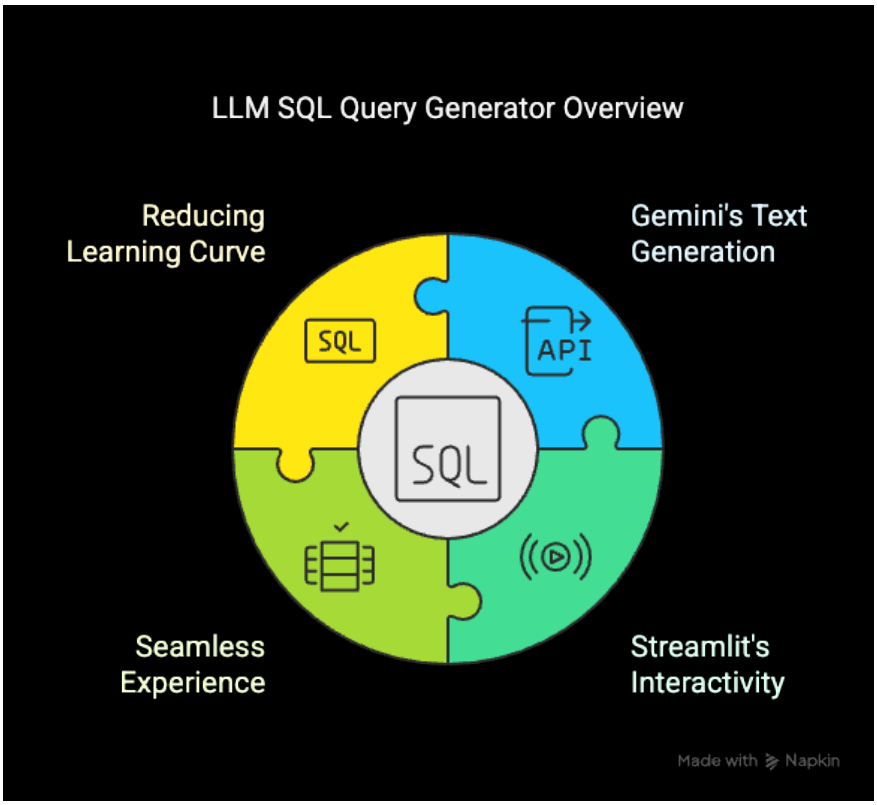
Query Translation Logic:

```
def get_gemini_response(question, prompt):
    model = genai.GenerativeModel('gemini-2.0-flash')
    response = model.generate_content([prompt, question])
    return response.text
```

This function takes:

- A structured prompt describing the database and examples.
- The user's question in natural language.
- **Query translation logic is applied on the basis of the accuracy level selected by the user.**

It returns a syntactically valid SQL query aligned with the PostgreSQL dialect.



2.4 Prompt Engineering

Effective LLM behavior hinges on high-quality prompts. Our prompt includes:

*** The schema of all relevant tables: student, companies, offers**
*** Column descriptions and data types**
*** A series of example query pairs (NL question → SQL) - Schema based tuning**
*** Clear directives for Gemini response**

Example Prompt Snippet:

- ```
  Schema:
  - student(student_id, name, cgpa)
  - companies(company_id, name, sector)
  - offers(student_id, company_id, package_lpa)

  Example 1:
  Q: List all students with CGPA > 9
  A: SELECT * FROM student WHERE cgpa > 9;

  Instructions:
  Respond with SQL only. Use correct PostgreSQL syntax.
  ```
  ```

2.5 SQL Execution via PostgreSQL

Once the SQL query is generated by the LLM, it is executed on a live PostgreSQL database using the `psycopg2` library.

SQL Execution Function:

```
def read_sql_query(sql):
    conn = psycopg2.connect(...)
    cursor = conn.cursor()
    cursor.execute(sql)
    result = cursor.fetchall()
    conn.close()
    return result
```

Post-query execution, the results are formatted and displayed on the web interface using:

```
st.table(result)
```

The result is displayed in pandas data frame format, based on the user input, completing the full loop from natural language to query result.

2.6 Custom Styling

To enhance usability and aesthetics, we add embedded CSS and Streamlit HTML components.

Styling Features:

- Gradient buttons with hover effects
- Sticky navigation bar for page switching
- Responsive layout for desktop and mobile views
- Highlighted results for better visibility

This visual polish contributes to a more professional and user-friendly interface.

3 Results and Demonstration

3.1 Sample Queries and Outputs

Along with the Generated SQL, the explanation of the applied logic is also displayed, when the user clicks on the "Show Explanation of the logic" button.
Below are some real examples to demonstrate the LLM's accuracy:

- Input: "List students with CGPA greater than 9"
  SQL: `SELECT * FROM student WHERE cgpa > 9;`
- Input: "Which company offered the highest salary?"
  SQL:

```
SELECT s.name, c.name, o.package_lpa
FROM student s
JOIN offers o ON s.student_id = o.student_id
JOIN companies c ON c.company_id = o.company_id
ORDER BY o.package_lpa DESC
LIMIT 1;
```

- Input: “Find students placed in Finance companies”
  SQL:

```
SELECT s.name
FROM student s
JOIN offers o ON s.student_id = o.student_id
JOIN companies c ON o.company_id = c.company_id
WHERE c.sector = 'Finance';
```

These examples validate the capability of the LLM to interpret diverse queries and generate precise outputs.

3.2 Interface Screenshots

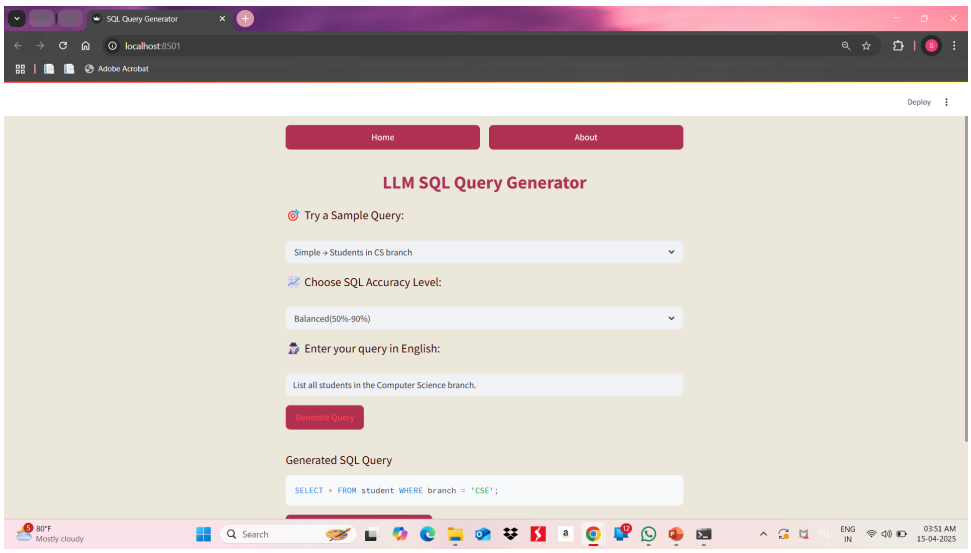


Figure 1: Execution of a query

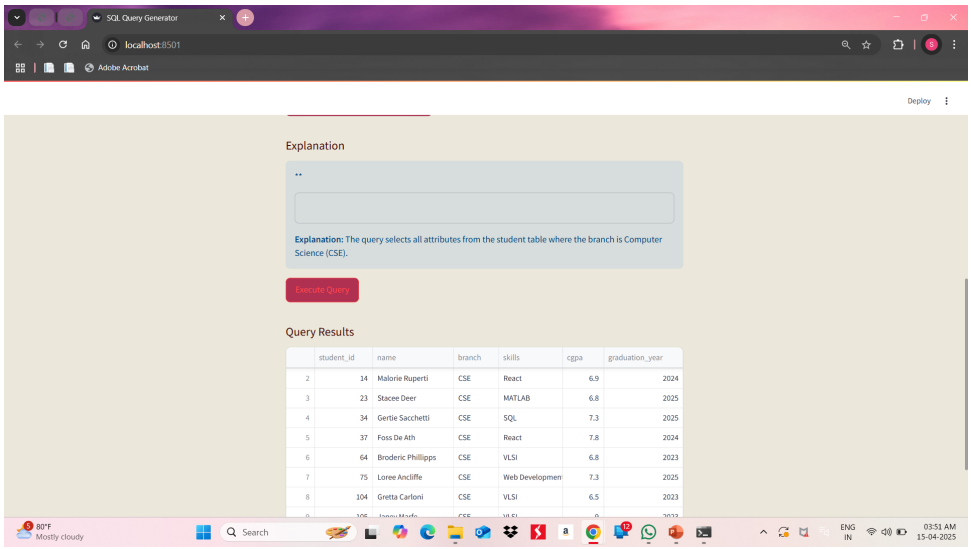


Figure 2: Output table generated from SQL result

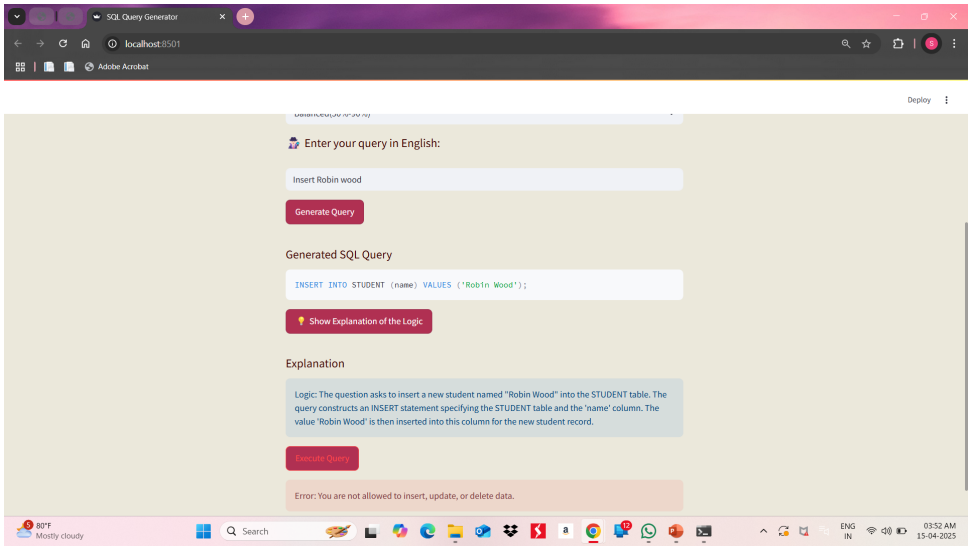


Figure 3: Error message when the user tries to insert data

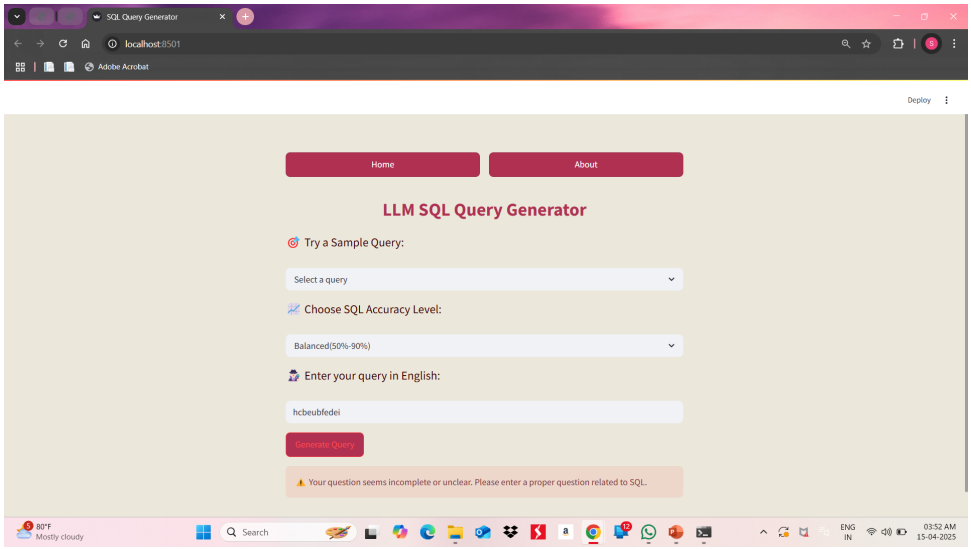


Figure 4: Error message for an incorrect query

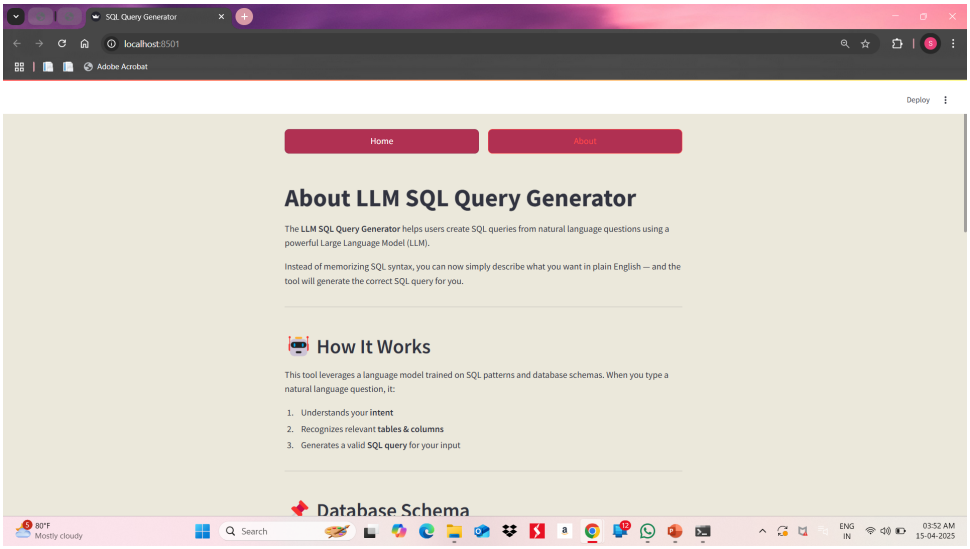


Figure 5: About page displaying the database schema

4 Conclusion and Future Scope

4.1 Conclusion

This project successfully demonstrates the seamless integration of LLMs and relational databases to create a low-barrier interface for data access. The core contributions include:

* **LLM-based translation of human language to SQL**
* **Real-time query execution on PostgreSQL**
* **Intuitive and responsive frontend using Streamlit**

It shows that with structured prompt engineering and careful backend integration, LLMs like Gemini can power practical tools for democratizing data.

4.2 Limitations

Despite promising results, certain limitations exist:

- Query Hallucinations: LLM may generate invalid or irrelevant SQL for ambiguous queries.
- Lack of Query Validation: No verification layer before query execution; dangerous commands may execute.
- Dependency on Prompt Quality: Translation accuracy is bound to how well the schema and examples are described.

4.3 Future Work

To improve robustness and expand usability, future extensions may include:

- Query sanitization and static analysis for safety
- Displaying SQL explanations alongside output
- Adding data visualizations (bar, pie charts)
- Supporting schema discovery and automatic table inference
- Role-based access and user authentication features

5 References

References

[1] Streamlit Documentation. https://docs.streamlit.io/

[2] Google Generative AI. https://cloud.google.com/generative-ai

[3] psycopg2 PostgreSQL Adapter. https://www.psycopg.org/docs/

[4] python-dotenv Library. https://pypi.org/project/python-dotenv/

[5] PostgreSQL Official Documentation. https://www.postgresql.org/docs/